CS:APP Chapter 4:
Processor Architecture*

Randal E. Bryant
David R. O'Hallaron

May 13, 2002

Contents

Modern microprocessors are among the most complex systems ever created by humans. A single silicon chip, roughly the size of a fingernail, can contain a complete, high-performance processor, large cache memories, and the logic required to interface it to external devices. In terms of performance, the processors implemented on a single chip today dwarf the room-sized supercomputers that cost over $10 million just 20 years ago. Even the embedded processors found in everyday appliances such as cell phones, personal digital assistants, and handheld game systems are far more powerful than the early developers of computers ever envisioned.

Thus far, we have only viewed computer systems down to the level of machine-language programs. We have seen that a processor must execute a sequence of instructions, where each instruction performs some primitive operation, such as adding two numbers. An instruction is encoded in binary form as a sequence of one or more bytes. The instructions supported by a particular processor and their byte-level encodings are known as its *instruction-set architecture* (ISA). Different "families" of processors, such as Intel IA32, IBM/Motorola PowerPC, and Sun Microsystems SPARC have different ISAs. A program compiled for one type of machine will not run on another. On the other hand, there are many different models of processors within a single family. Each manufacturer produces processors of ever-growing performance and complexity, but the different models remain compatible at the ISA level. Popular families, such as IA32, have processors supplied by multiple manufacturers. Thus, the ISA provides a conceptual layer of abstraction between compiler writers, who need only know what instructions are permitted and how they are encoded, and processor designers, who must build machines that execute those instructions.

In this chapter, we take a brief look at the design of processor hardware. We study the way a hardware system can execute the instructions of a particular ISA. This view will give you a better understanding of how computers work and the technological challenges faced by computer manufacturers. One important concept is that the actual way a modern processor operates can be quite different from the model of computation implied by the ISA. The ISA model would seem to imply *sequential* instruction execution, where each instruction is fetched and executed to completion before the next one begins. By executing different parts of multiple instructions simultaneously, the processor can achieve higher performance than if it executed just one instruction at a time. Special mechanisms are used to make sure the processor computes the same results as it would with sequential execution. This idea of using clever tricks to improve performance while maintaining the functionality of a simpler and more abstract model is well known in computer science. Examples include the use of caching in Web browsers and information retrieval data structures such as balanced binary trees and hash tables.

Chances are you will never design your own processor. This is a task for experts working at fewer than 100 companies worldwide. Why, then, should you learn about processor design?

- *It is intellectually interesting.* There is an intrinsic value in learning how things work. It is especially interesting to learn the inner workings of a system that is such a part of the daily lives of computer scientists and engineers and yet remains a mystery to many. Processor design embodies many of the principles of good engineering practice. It requires creating as simple a structure as possible to perform a complex task.
- *Understanding how the processor works aids in understanding how the overall computer system works.* In Chapter **??**, we will look at the memory system and the techniques used to create an image of a very large memory with a very fast access time. Seeing the processor side of the processor-memory interface will make this presentation more complete.

- *Although few people design processors, many design hardware systems containing processors.* This has become commonplace as processors are embedded into real-world systems such as automobiles and appliances. Embedded system designers must understand how processors work, because these systems are generally designed and programmed at a lower level of abstraction than is the case for desktop systems.

- *You just might work on a processor design.* Although the number of companies producing microprocessors is small, the design teams working on those processors are already large and growing. There can be over 800 people involved in the different aspects of a major processor design.

In this chapter, we start by defining a simple instruction set that we use as a running example for our processor implementations. We call this the "Y86" instruction set, because it was inspired by the IA32 instruction set, which is colloquially referred to as "X86." Compared with IA32, the Y86 instruction set has fewer data types, instructions, and addressing modes. It also has a simpler byte-level encoding. Still, it is sufficiently complete to allow us to write simple programs manipulating integer data. Designing a processor to implement Y86 requires us to face many of the challenges faced by processor designers.

We then provide some background on digital hardware design. We describe the basic building blocks used in a processor and how they are connected together and operated. This presentation builds on our discussion of Boolean algebra and bit-level operations from Chapter **??**. We also introduce a simple language, HCL (for "Hardware Control Language") to describe the control portions of hardware systems. We will later use this language to describe our processor designs. Even if you already have some background in logic design, read this section to understand our particular notation.

As a first step in designing a processor, we present a functionally correct, but somewhat impractical, Y86 processor based on *sequential* operation. This processor executes a complete Y86 instruction on every clock cycle. The clock must run slowly enough to allow an entire series of actions to complete within one cycle. Such a processor could be implemented, but its performance would be well below what could be achieved for this much hardware.

With the sequential design as a basis, we then apply a series of transformations to create a *pipelined* processor. This processor breaks the execution of each instruction into five steps, each of which is handled by a separate section or *stage* of the hardware. Instructions progress through the stages of the pipeline, with one instruction entering the pipeline on each clock cycle. As a result, the processor can be executing the different steps of up to five instructions simultaneously. Making this processor preserve the sequential behavior of the Y86 ISA requires handling a variety of *hazard* conditions, where the location or operands of one instruction depend on those of other instructions that are still in the pipeline.

We have devised a variety of tools for studying and experimenting with our processor designs. These include an assembler for Y86, a simulator for running Y86 programs on your machine, and simulators for two sequential and one pipelined processor design. The control logic for these designs is described by files in HCL notation. By editting these files and recompiling the simulator, you can alter and extend the simulation behavior. A number of exercises are provided that involve implementing new instructions and modifying how the machine processes instructions. Testing code is provided to help you evaluate the correctness of your modifications. These exercises will greatly aid your understanding of the material and will give you an appreciation for the many different design alternatives faced by processor designers.

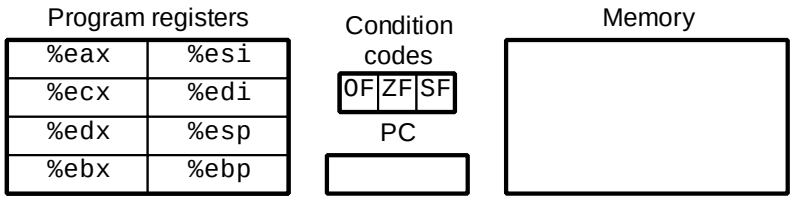


Figure 1: **Y86 programmer-visible state.** As with IA32, programs for Y86 access and modify the program registers, the condition code, the program counter (PC), and the memory.

1 The Y86 Instruction Set Architecture

As Figure 1 illustrates, each instruction in a Y86 program can read and modify some part of the processor state. This is referred to as the *programmer-visible* state, where the "programmer" in this case is either someone writing programs in assembly code or a compiler generating machine-level code. We will see in our processor implementations that we do not need to represent and organize this state in exactly the manner implied by the ISA, as long as we can make sure that machine-level programs appear to have access to the programmer-visible state. The state for Y86 is similar to that for IA32. There are eight *program registers*: `%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`, `%edi`, `%esp`, and `%ebp`. Each of these stores a word. Register `%esp` is used as a stack pointer by the push, pop, call, and return instructions. Otherwise, the registers have no fixed meanings or values. There are three single-bit *condition codes*: `ZF`, `SF`, and `OF`, storing information about the effect of the most recent arithmetic or logical instruction. The program counter (PC) holds the address of the instruction currently being executed. The *memory* is conceptually a large array of bytes, holding both program and data. Y86 programs reference memory locations using *virtual addresses*. A combination of hardware and operating system software translates these into the actual, or *physical* addresses indicating where the values are actually stored in memory. We will study virtual memory in more detail in Chapter **??**. For now, we can think of the virtual memory system as providing Y86 programs with an image of a monolithic byte array.

Figure 2 gives a concise description of the individual instructions in the Y86 ISA. We use this instruction set as a target for our processor implementations. The set of Y86 instructions is largely a subset of the IA32 instruction set. It includes only four-byte integer operations; it has fewer addressing modes; and it includes a smaller set of operations. Since we only use four-byte data, we refer to these as "words." In this figure, we show the assembly code representation of the instructions on the left and the byte encodings on the right. The assembly code is similar to the GAS representation of IA32 programs.

Here are some further details about the different Y86 instructions.

- The IA32 `movl` instruction is split into four different instructions: `irmovl`, `rrmovl`, `mrmovl`, and `rmmovl`, explicitly indicating the form of the source and destination. The source is either immediate (`i`), register (`r`), or memory (`m`). It is designated by the first character in the instruction name. The destination is either register (`r`) or memory (`m`). It is designated by the second character in the instruction name. Explicitly identifying the four types of data transfer will prove helpful when we decide how to implement them.

  The memory references for the two memory movement instructions have a simple base and displace-

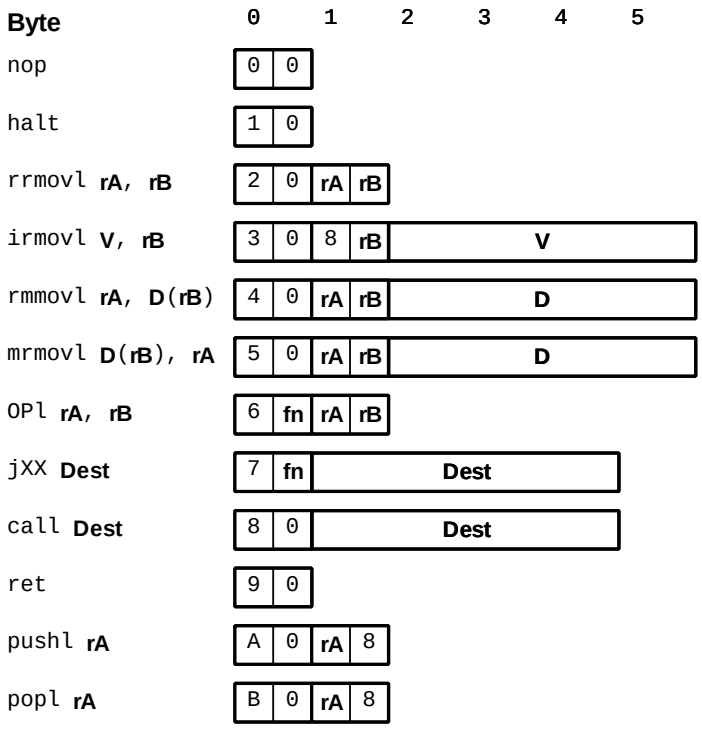


Figure 2: **Y86 instruction set.** Instruction encodings range between 1 and 6 bytes. An instruction consists of a one-byte instruction specifier, possibly a one-byte register specifier, and possibly a four-byte constant word. Field `fn` specifies a particular integer operation (`OPl`) or a particular branch condition (`jXX`). All numeric values are shown in hexadecimal.

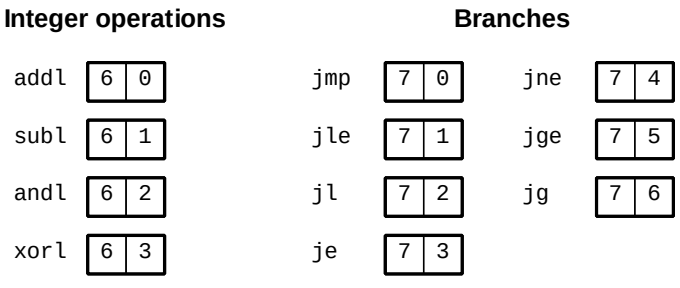


Figure 3: **Function codes for Y86 instruction set.** The codes specify a particular integer operation or branch condition. These instructions are shown as `OPl` and `jXX` in Figure 2.

Number	Register name
0	%eax
1	%ecx
2	%edx
3	%ebx
6	%esi
7	%edi
4	%esp
5	%ebp
8	No register

Figure 4: **Y86 program register identifiers.** Each of the eight program registers has an associated identifer (ID) ranging from 0 to 7. ID 8 in a register field of an instruction indicates the absence of a register operand.

ment format. We do not support the second index register or any scaling of the register value in the address computation.

As with IA32, we do not allow direct transfers from one memory location to another. In addition, we do not allow a transfer of immediate data to memory.

- There are four integer operation instructions, shown in Figure 2 as `OPl`. These are `addl`, `subl`, `andl`, and `xorl`. They operate only on register data, whereas IA32 also allows operations on memory data. These instructions set the three condition codes `ZF`, `SF`, and `OF` (zero, sign, and overflow).
- The seven jump instructions (shown in Figure 2 as `jXX`) are `jmp`, `jle`, `jl`, `je`, `jne`, `jge`, and `jg`. Branches are taken according to the type of branch and the settings of the condition codes. The branch conditions are the same as with IA32 (Figure **??**).
- The `call` instruction pushes the return address on the stack and jumps to the destination address. The `ret` instruction returns from such a call.
- The `pushl` and `popl` instructions implement push and pop, just as they do in IA32.
- The `halt` instruction stops instruction execution. IA32 has a comparable instruction, called `hlt`. IA32 application programs are not permitted to use this instruction, since it causes the entire system to stop. We use `halt` in our Y86 programs to stop the simulator.

Figure 2 also shows the byte-level encoding of the instructions. Each instruction requires between one and six bytes, depending on which fields are required. Every instruction has an initial byte identifying the instruction type. This byte is split into two four-bit parts: the high-order or *code* part, and the low-order or *function* part. As you can see in Figure 2, code values range from `0` to hexadecimal `B`. The function values are significant only for the cases where a group of related instructions share a common code. These are given in Figure 3, showing the specific encodings of the integer operation and branch instructions.

As shown in Figure 4, each of the eight program registers has an associated *register identifier* (ID) ranging from 0 to 7. The numbering of registers in Y86 matches what is used in IA32. The program registers are

stored within the CPU in a *register file*, a small random-access memory where the register IDs serve as addresses. ID value 8 is used in the instruction encodings and within our hardware designs when we need to indicate that no register should be accessed.

Some instructions are just one byte long, but those that require operands have longer encodings. First, there can be an additional *register specifier byte*, specifying either one or two registers. These register fields are called rA and rB in Figure 2. As the assembly code versions of the instructions show, they can specify the registers used for data sources and destinations, as well as the base register used in an address computation, depending on the instruction type. Instructions that have no register operands, such as branches and `call`, do not have a register specifier byte. Those that require just one register operand (`irmovl`, `pushl`, and `popl`) have the other register specifier set to value 8. This convention will prove useful in our processor implementation.

Some instructions require an additional four-byte *constant word*. This word can serve as the immediate data for `irmovl`, the displacement for `rmmovl` and `mrmovl` address specifiers, and the destination of branches and calls. Note that branch and call destinations are given as absolute addresses, rather than using the PC-relative addressing seen in IA32. Processors use PC-relative addressing to give more compact encodings of branch instructions and to allow code to be copied from one part of memory to another without the need to update all of the branch target addresses. Since we are more concerned with simplicity in our presentation, we use absolute addressing. As with IA32, all integers have a little-endian encoding. When the instruction is written in disassembled form, these bytes appear in reverse order.

As an example, let us generate the byte encoding of the instruction `rmmovl %esp,0x12345(%edx)` in hexadecimal. From Figure 2 we can see that `rmmovl` has initial byte `40`. We can also see that source register `%esp` should be encoded in the rA field, and base register `%edx` should be encoded in the rB field. Using the register numbers in Figure 4, we get a register specifier byte of `42`. Finally, the displacement is encoded in the four-byte constant word. We first pad `0x12345` with leading 0s to fill out four bytes, giving a byte sequence of `00 01 23 45`. We write this in byte-reversed order as `45 23 01 00`. Combining these we get an instruction encoding of `404245230100`.

One important property of any instruction set is that the byte encodings must have a unique interpretation. An arbitrary sequence of bytes either encodes a unique instruction sequence or is not a legal byte sequence. This property holds for Y86, because every instruction has a unique combination of code and function in its initial byte, and given this byte, we can determine the length and meaning of any additional bytes. This property ensures that a processor can execute an object code program without any ambiguity about the meaning of the code. Even if the code is embedded within other bytes in the program, we can readily determine the instruction sequence as long as we start from the first byte in the sequence. On the other hand, if we do not know the starting position of a code sequence, we cannot reliably determine how to split the sequence into individual instructions. This causes problems for disassemblers and other tools that attempt to extract machine-level programs directly from object code byte sequences.

Practice Problem 1:

Determine the byte encoding of the Y86 instruction sequence that follows. The line "`.pos 0x100`" indicates that the starting address of the object code should be `0x100`.

```
.pos 0x100  # Start generating code at address 0x100
    irmovl $15,%ebx
```

```
    rrmovl %ebx,%ecx
loop:
    rmmovl %ecx,-3(%ebx)
    addl   %ebx,%ecx
    jmp loop
```

Practice Problem 2:

For each byte sequence listed, determine the Y86 instruction sequence it encodes. If there is some invalid byte in the sequence, show the instruction sequence up to that point and indicate where the invalid value occurs. For each sequence, we show the starting address, then a colon, and then the byte sequence.

A. `0x100:3083fcffffff40630008000010`
B. `0x200:a06880080200001030830a00000090`
C. `0x300:50540700000000f0b018`
D. `0x400:6113730004000010`
E. `0x500:6362a080`

Aside: Comparing IA32 to Y86 Instruction Encodings
Compared with the instruction encodings used in IA32, the encoding of Y86 is much simpler but also less compact. The register fields only occur in fixed positions in all Y86 instructions, whereas they are packed into various positions in the different IA32 instructions. We use a four-bit encoding of registers, even though there are only eight possible registers. IA32 uses just 3 bits. Thus, IA32 can pack a push or pop instruction into just one byte, with a 5-bit field indicating the instruction type and the remaining 3 bits for the register specifier. IA32 can encode constant values in 1, 2, or 4 bytes, whereas Y86 always requires 4 bytes. **End Aside.**

Aside: RISC and CISC Instruction Sets
IA32 is sometimes labeled as a "complex instruction set computer" (CISC—pronounced "sisk"), and is deemed to be the opposite of ISAs that are classified as "reduced instruction set computers" (RISC—pronounced "risk"). Historically, CISC machines came first, having evolved from the earliest computers. By the early 1980s, instruction sets for mainframe and minicomputers had grown quite large, as machine designers incorporated new instructions to support high-level tasks, such as manipulating circular buffers, peforming decimal arithmetic, and evaluating polynomials. The first microprocessors appeared in the early 1970s and had limited instruction sets, because the integrated circuit technology then posed severe constraints on what could be implemented on a single chip. Microprocessors evolved quickly and, by the early 1980s, were following the path of increasing instruction-set complexity set by mainframes and minicomputers. The 80x86 family took this path, evolving into IA32. Even IA32 continues to evolve as new classes of instructions are added to support the processing required by multimedia applications.

The RISC design philosophy developed in the early 1980s as an alternative to these trends. A group of hardware and compiler experts at IBM, strongly influenced by the ideas of IBM researcher John Cocke, recognized that they could generate efficient code for a much simpler form of instruction set. In fact, many of the high-level instructions that were being added to instruction sets were very difficult to generate with a compiler and were seldom used. A simpler instruction set could be implemented with much less hardware and could be organized in an efficient pipeline structure, similar to those described later in this chapter. IBM did not commercialize this idea until many years later, when it developed the Power and PowerPC ISAs.

The RISC concept was further developed by Professors David Patterson, of the University of California at Berkeley, and John Hennessy, of Stanford University. Patterson gave the name RISC to this new class of machines, and CISC to the existing class, since there had previously been no need to have a special designation for a nearly universal form of instruction set.

Comparing CISC with the original RISC instruction sets, we find the following general characteristics:

CISC	Early RISC
A large number of instructions. The Intel document describing the complete set of instructions [1] is over 700 pages long.	Many fewer instructions. Typically less than 100.
Some instructions with long execution times. These include instructions that copy an entire block from one part of memory to another and others that copy multiple registers to and from memory.	No instruction with a long execution time. Some early RISC machines did not even have an integer multiply instruction, requiring compilers to implement multiplication as a sequence of additions.
Variable-length encodings. IA32 instructions can range from 1 to 15 bytes.	Fixed length encodings. Typically all instructions are encoded as four bytes.
Multiple formats for specifying operands. In IA32, a memory operand specifier can have many different combinations of displacement, base and index registers, and scale factors.	Simple addressing formats. Typically just base and displacment addressing.
Arithmetic and logical operations can be applied to both memory and register operands.	Arithmetic and logial operations only use register operands. Memory referencing is only allowed by *load* instructions, reading from memory into a register, and *store* instructions, writing from a register to memory. This convention is referred to as a *load/store architecture*.
Implementation artifacts hidden from machine-level programs. The ISA provides a clean abstraction between programs and how they get executed.	Implementation artifacts exposed to machine-level programs. Some RISC machines prohibit particular instruction sequences and have jumps that do not take effect until the following instruction is executed. The compiler is given the task of optimizing performance within these constraints.
Condition codes. Special flags are set as a side effect of instructions and then used for conditional branch testing.	No condition codes. Instead, explicit test instructions that store the test result in a normal register are used for conditional evaluation.
Stack-intensive procedure linkage. The stack is used for procedure arguments and return addresses.	Register-intensive procedure linkage. Registers are used for procedure arguments and return addressess. Some procedures can thereby avoid any memory references. Typically, the processor has many more (up to 32) registers.

The Y86 instruction set includes attributes of both CISC and RISC instruction sets. On the CISC side, it has condition codes, variable-length instructions, and stack-intensive procedure linkages. On the RISC side, it uses a load-store architecture and a regular encoding. It can be viewed as taking a CISC instruction set (IA32) and simplifying it by applying some of the principles of RISC. **End Aside.**

Aside: The RISC Versus CISC Controversy

Through the 1980s, battles raged in the computer architecture community regarding the merits of RISC versus CISC instruction sets. Proponents of RISC claimed they could get more computing power for a given amount of hardware through a combination of streamlined instruction set design, advanced compiler technology, and pipelined processor implementation. CISC proponents countered that fewer CISC instruction were required to perform a given task, and so their machines could achieve higher overall performance.

Major companies introduced RISC processor lines, including Sun Microsystems (SPARC), IBM and Motorola (PowerPC), and Digital Equipment Corporation (Alpha).

In the early 1990s, the debate diminished as it became clear that neither RISC nor CISC in their purest forms were better than designs that incorporated the best approaches of both. RISC machines evolved and introduced more instructions, many of which take multiple cycles to execute. RISC machines today have hundreds of instructions in their repertoire, hardly fitting the name “reduced instruction set machine.” The idea of exposing implementation

IA32 code

```
     int Sum(int *Start, int Count)
 1 Sum:
 2   pushl %ebp
 3   movl %esp,%ebp
 4   movl 8(%ebp),%ecx    ecx = Start
 5   movl 12(%ebp),%edx   edx = Count
 6   xorl %eax,%eax       sum = 0
 7   testl %edx,%edx
 8   je .L34
 9 .L35:
10   addl (%ecx),%eax     add *Start to sum
11   addl $4,%ecx         Start++
12   decl %edx            Count--
13   jnz .L35             Stop when 0
14 .L34:
15   movl %ebp,%esp
16   popl %ebp
17   ret
```

Y86 code

```
     int Sum(int *Start, int Count)
 1 Sum:    pushl %ebp
 2   rrmovl %esp,%ebp
 3   mrmovl 8(%ebp),%ecx       ecx = Start
 4   mrmovl 12(%ebp),%edx      edx = Count
 5   irmovl $0, %eax           sum = 0
 6   andl   %edx,%edx
 7   je     End
 8 Loop:   mr-
movl (%ecx),%esi        get *Start
 9   addl %esi,%eax            add to sum
10   irmovl $4,%ebx
11   addl %ebx,%ecx            Start++
12   irmovl $-1,%ebx
13   addl %ebx,%edx            Count-
                               -
14   jne    Loop               Stop when 0
15 End:
16   popl %ebp
17   ret
```

Figure 5: **Comparison of Y86 and IA32 assembly programs.** The Sum function computes the sum of an integer array. The Y86 code differs from the IA32 mainly in that it may require multiple instructions to perform what can be done with a single IA32 instruction.

artifacts to machine-level programs proved to be short-sighted. As new processor models were developed using more advanced hardware structures, many of these artifacts became irrelevant, but they still remained part of the instruction set. Still, the core of RISC design is an instruction set that is well-suited to execution on a pipelined machine.

More recent CISC machines also take advantage of high-performance pipeline structures. As we will discuss in Section **??**, they fetch the CISC instructions and dynamically translate them into a sequence of simpler, RISC-like operations. For example, an instruction that adds a register to memory is translated into three operations: one to read the original memory value, one to perform the addition, and a third to write the sum to memory. Since the dynamic translation can generally be performed well in advance of the actual instruction execution, the processor can sustain a very high execution rate.

Marketing issues, apart from technological ones, have also played a major role in determining the success of different instruction sets. By maintaining compatibility with its existing processors, Intel with IA32 made it easy to keep moving from one generation of processor to the next. As integrated circuit technology improved, Intel and other IA32 processor manufacturers could overcome the inefficiencies created by the original 8086 instruction-set design, using RISC techniques to produce performance comparable to the best RISC machines. In the areas of desktop and laptop computing, IA32 has achieved total domination.

RISC processors have done very well in the market for *embedded processors*, controlling such systems as cellular telephones, automobile brakes, and Internet appliances. In these applications, saving on cost and power is more important than maintaining backward compatibility. In terms of the number of processors sold, this is a very large and growing market. **End Aside.**

Figure 5 shows IA32 and Y86 assembly code for the following C function:

```
int Sum(int *Start, int Count)
{
  int sum = 0;
  while (Count) {
    sum += *Start;
    Start++;
    Count--;
  }
  return sum;
}
```

The IA32 code was generated by C compiler GCC. The Y86 code is essentially the same, except that Y86 sometimes requires two instructions to accomplish what can be done with a single IA32 instruction. If we had written the program using array indexing, however, the conversion to Y86 code would be more difficult, since Y86 does not have scaled addressing modes.

Figure 6 shows an example of a complete program file written in Y86 assembly code. The program contains both data and instructions. Directives indicate where to place code or data and how to align it. The program specifies issues such as stack placement, data initialization, program initialization, and program termination.

In this program, words beginning with "`.`" are *assembler directives* telling the assembler to adjust the address at which it is generating code or to insert some words of data. The directive `.pos 0` (line 2) indicates that the assembler should begin generating code starting at address 0. This is the starting point of all Y86 programs. The next two instructions (lines 3 and 4) initialize the stack and frame pointers. We can see that the label `Stack` is declared at the end of the program (line 39), to indicate address `0x100` using a `.pos` directive (line 38). Our stack will therefore start at this address and grow downward.

Lines 8 to 12 of the program declare an array of four words, having values `0xd`, `0xc0`, `0xb00`, and `0xa000`. The label `array` denotes the start of this array, and is aligned on a four-byte boundary (using the `.align` directive). Lines 14 to 19 show a "main" procedure that calls the function `Sum` on the four-word array and then halts.

As this example shows, writing a program in Y86 requires the programmer to perform tasks we ordinarily assign to the compiler, linker, and runtime system. Fortunately, we only do this for small programs for which simple mechanisms suffice.

Figure 7 shows the result of assembling the code shown in Figure 6 by an assembler we call YAS. The assembler output is in ASCII format to make it more readable. On lines of the assembly file that contain instructions or data, the object code contains an address, followed by the values of between 1 and 6 bytes.

We have implemented an instruction set simulator we call YIS. Running on our sample object code, it generates the following output:

```
Stopped in 46 steps at PC = 0x3a.  Exception 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x0000abcd
%ecx:   0x00000000      0x00000024
%ebx:   0x00000000      0xffffffff
%esp:   0x00000000      0x000000f8
%ebp:   0x00000000      0x00000100
```

code/arch/y86-code/asum.ys

```
1  # Execution begins at address 0
2          .pos 0
3  init:   irmovl Stack, %esp      # Set up Stack pointer
4          irmovl Stack, %ebp      # Set up base pointer
5          jmp Main                # Execute main program
6
7  # Array of 4 elements
8          .align 4
9  array:  .long 0xd
10         .long 0xc0
11         .long 0xb00
12         .long 0xa000
13
14 Main:   irmovl $4,%eax
15         pushl %eax      # Push 4
16         irmovl array,%edx
17         pushl %edx      # Push array
18         call Sum        # Sum(array, 4)
19         halt
20
21         # int Sum(int *Start, int Count)
22 Sum:    pushl %ebp
23         rrmovl %esp,%ebp
24         mrmovl 8(%ebp),%ecx     # ecx = Start
25         mrmovl 12(%ebp),%edx    # edx = Count
26         irmovl $0, %eax         # sum = 0
27         andl   %edx,%edx
28         je     End
29 Loop:   mrmovl (%ecx),%esi      # get *Start
30         addl %esi,%eax          # add to sum
31         irmovl $4,%ebx          #
32         addl %ebx,%ecx          # Start++
33         irmovl $-1,%ebx         #
34         addl %ebx,%edx          # Count--
35         jne    Loop             # Stop when 0
36 End:
37         popl %ebp
38         ret
39         .pos 0x100
40 Stack:  # The stack goes here
```

code/arch/y86-code/asum.ys

Figure 6: **Sample program written in Y86 assembly code.** The Sum function is called to compute the sum of a 4-element array.

code/arch/y86-code/asum.yo

```
                    | # Execution begins at address 0
  0x000:            |         .pos 0
  0x000: 308400010000 | init:   irmovl Stack, %esp      # Set up Stack pointer
  0x006: 308500010000 |         irmovl Stack, %ebp      # Set up base pointer
  0x00c: 7024000000   |         jmp Main                # Execute main program
                    |
                    | # Array of 4 elements
  0x014:            |         .align 4
  0x014: 0d000000     | array:  .long 0xd
  0x018: c0000000     |         .long 0xc0
  0x01c: 000b0000     |         .long 0xb00
  0x020: 00a00000     |         .long 0xa000
                    |
  0x024: 308004000000 | Main:   irmovl $4,%eax
  0x02a: a008         |         pushl %eax      # Push 4
  0x02c: 308214000000 |         irmovl array,%edx
  0x032: a028         |         pushl %edx      # Push array
  0x034: 803a000000   |         call Sum        # Sum(array, 4)
  0x039: 10           |         halt
                    |
                    |         # int Sum(int *Start, int Count)
  0x03a: a058         | Sum:    pushl %ebp
  0x03c: 2045         |         rrmovl %esp,%ebp
  0x03e: 501508000000 |         mrmovl 8(%ebp),%ecx     # ecx = Start
  0x044: 50250c000000 |         mrmovl 12(%ebp),%edx    # edx = Count
  0x04a: 308000000000 |         irmovl $0, %eax         # sum = 0
  0x050: 6222         |         andl   %edx,%edx
  0x052: 7374000000   |         je     End
  0x057: 506100000000 | Loop:   mrmovl (%ecx),%esi      # get *Start
  0x05d: 6060         |         addl %esi,%eax          # add to sum
  0x05f: 308304000000 |         irmovl $4,%ebx          #
  0x065: 6031         |         addl %ebx,%ecx          # Start++
  0x067: 3083ffffffff |         irmovl $-1,%ebx         #
  0x06d: 6032         |         addl %ebx,%edx          # Count--
  0x06f: 7457000000   |         jne    Loop             # Stop when 0
  0x074:            | End:
  0x074: b058         |         popl %ebp
  0x076: 90           |         ret
  0x100:            |         .pos 0x100
  0x100:            | Stack:  # The stack goes here
```

code/arch/y86-code/asum.yo

Figure 7: **Output of YAS assembler.** Each line includes a hexadecimal address and between 1 and 6 bytes of object code.

```
%esi:   0x00000000      0x0000a000

Changes to memory:
0x00f0: 0x00000000      0x00000100
0x00f4: 0x00000000      0x00000039
0x00f8: 0x00000000      0x00000014
0x00fc: 0x00000000      0x00000004
```

The simulator only prints out words that change during simulation, either in registers or in memory. The original values (here they are all 0) are shown on the left, and the final values are shown on the right. We can see in this output that register `%eax` contains `0xabcd`, the sum of the four-element array passed to subroutine `Sum`. In addition, we can see that the stack, which starts at address `0x100` and grows downward, has been used, causing changes to memory at addresses `0xf0` through `0xfc`.

Practice Problem 3:

Write Y86 code to implement a recursive sum function `rSum`, based on the following C code:

```
int rSum(int *Start, int Count)
{
    if (Count <= 0)
        return 0;
    return *Start + rSum(Start+1, Count-1);
}
```

You might find it helpful to compile the C code on an IA32 machine and then translate the instructions to Y86.

Practice Problem 4:

The `pushl` instruction both decrements the stack pointer by 4 and writes a register value to memory. It is not totally clear what the processor should do with the instruction `pushl %esp`, since the register being pushed is being changed by the same instruction. Two conventions are possible: (1) push the original value of `%esp`, or (2) push the decremented value of `%esp`.

Let's resolve this issue by doing the same thing an IA32 processor would do. We could try reading the Intel documentation on this instruction, but a simpler approach is to conduct an experiment on an actual machine. The C compiler would not normally generate this instruction, so we must use hand-generated assembly code for this task. As described in Section **??**, the best way to insert small amounts of assembly code into a C program is to use the `asm` feature of GCC. Here is a test program we have written. Rather than attempting to read the `asm` declaration, you will find it easiest to read the assembly code in the comment preceding it.

```
int pushtest()
{
    int rval;
    /* Insert the following assembly code:
       movl %esp,%eax   # Save stack pointer
```

```
       pushl %esp       # Push stack pointer
       popl  %edx       # Pop it back
       subl  %edx,%eax  # 0 or 4
       movl  %eax,rval  # Set as return value
    */
  asm("movl %%esp,%%eax;pushl %%esp;popl %%edx;subl %%edx,%%eax;movl %%eax,%0"
       : "=r" (rval)
       : /* No Input */
       : "%edx", "%eax");
    return rval;
}
```

In our experiments, we find that the function `pushtest` returns 0. What does this imply about the behavior of the instruction `pushl %esp` under IA32?

Practice Problem 5:

A similar ambiguity occurs for the instruction `popl %esp`. It could either set `%esp` to the value read from memory or to the incremented stack pointer. As with practice problem 4, let us run an experiment to determine how an IA32 machine would handle this instruction and then design our Y86 machine to follow the same convention.

```
int poptest(int tval)
{
    int rval;
    /* Insert the following assembly code:
       pushl tval     # Save tval on stack
       movl %esp,%edx # Save stack pointer
       popl %esp      # Pop to stack pointer.
       movl %esp,rval # Set popped value as return value
       movl %edx,%esp # Restore original stack pointer
    */
    asm("pushl %1; movl %%esp,%%edx; popl %%esp; movl %%esp,%0; movl %%edx,%%esp"
        : "=r" (rval)
        : "r" (tval)
        : "%edx");
    return rval;
}
```

We find this function always returns `tval`, the value passed to it as its argument. What does this imply about the behavior of `popl %esp`? What other Y86 instruction would have the exact same behavior?

2 Logic Design and the Hardware Control Language HCL

In hardware design, electronic circuits are used to compute functions on bits and to store bits in different kinds of memory elements. Most contemporary circuit technology represents different bit values as high or

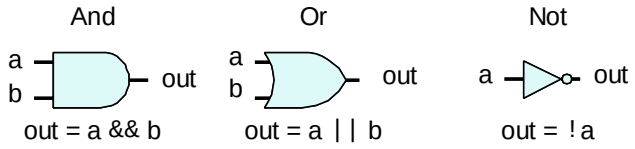


Figure 8: **Logic gate types.** Each gate generates output equal to some Boolean function of its inputs.

low voltages on signal wires. In current technology, logic value 1 is represented by a high voltage of around 1.0 volt, while logic value 0 is represented by a low voltage of around 0.0 volts. Three major components are required to implement a digital system: combinational logic to compute functions on the bits, memory elements to store bits, and clock signals to regulate the updating of the memory elements.

In this section, we provide a brief description of these different components. We also introduce HCL (for "hardware control language"), the language that we use to describe the control logic of the different processor designs. We only describe HCL informally here. A complete reference for HCL can be found in Appendix 6.1.

Aside: Modern Logic Design
At one time hardware designers created circuit designs by drawing schematic diagrams of logic circuits (first with paper and pencil and later with computer graphics terminals). Nowadays, most designs are expressed in a *hardware description language* (HDL), a textual notation that looks similar to a programming language but that is used to describe hardware structures rather than program behaviors. The most commonly used languages are Verilog, having a syntax similar to C, and VHDL, having a syntax similar to the Ada programming language. These languages were originally designed for expressing simulation models of digital circuits. In the mid-1980s, researchers developed *logic synthesis* programs that could generate efficient circuit designs from HDL descriptions. There are now a number of commercial synthesis programs, and this has become the dominant technique for generating digital circuits. This shift from hand-designed circuits to synthesized ones can be likened to the shift from writing programs in assembly code to writing them in a high-level language and having a compiler generate the machine code. **End Aside.**

2.1 Logic Gates

Logic gates are the basic computing elements for digital circuits. They generate an output equal to some Boolean function of the bit values at their inputs. Figure 8 shows the standard symbols used for Boolean functions AND, OR, and NOT. HCL expressions are shown below the gates for the Boolean operations. As you can see, we adopt the syntax for logic operators in C (Section **??**): && for AND, || for OR, and ! for NOT. We use these instead of the bit-level C operators &, |, and ~, because logic gates operate only on single-bit quantities, not entire words.

Logic gates are always active. If some input to a gate changes, then within some small amount of time, the output will change accordingly.

2.2 Combinational Circuits and HCL Boolean Expressions

By assembling a number of logic gates into a network, we can construct computational blocks known as *combinational circuits*. Two restrictions are placed on how the networks are constructed:

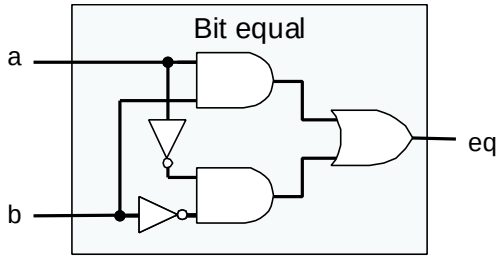


Figure 9: **Combinational circuit to test for bit equality.** The output will equal 1 when both inputs are 0, or both are 1.

- The outputs of two or more logic gates cannot be connected together. Otherwise the two could try to drive the wire in opposite directions, possibly causing an invalid voltage or a circuit malfunction.
- The network must be *acyclic*. That is, there cannot be a path through a series of gates that forms a loop in the network. Such loops can cause ambiguity in the function computed by the network.

Figure 9 shows an example of a simple combinational circuit that we will find useful. It has two inputs, a and b. It generates a single output eq, such that the output will equal 1 if either a and b are both 1 (detected by the upper AND gate) or are both 0 (detected by the lower AND gate). We write the function of this network in HCL as:

```
bool eq = (a && b) || (!a && !b);
```

This code simply defines the bit-level (denoted by data type `bool`) signal eq as a function of inputs a and b. As this example shows HCL uses C-style syntax, with '=' associating a signal name with an expression. Unlike C, however, we do not view this as performing a computation and assigning the result to some memory location. Instead, it is simply a way to give a name to an expression.

> **Practice Problem 6**:
>
> Write an HCL expression for a signal xor, equal to the EXCLUSIVE-OR of inputs a and b. What is the relation between the signals xor and eq defined above?

Figure 10 shows another example of a simple but useful combinational circuit known as a *multiplexor*. A multiplexor selects a value from among a set of different data signals, depending on the value of a control input signal. In this single-bit multiplexor, the two data signals are the input bits a and b, while the control signal is the input bit s. The output will equal a when s is 1, and it will equal b when s is 0. In this circuit, we can see that the two AND gates determine whether to pass their respective data inputs to the OR gate. The upper AND gate passes signal b when s is 0 (since the other input to the gate is !s), while the lower AND gate passes signal a when s is 1. Again, we can write an HCL expression for the output signal, using the same operations as are present in the combinational circuit:

```
bool out = (s && a) || (!s && b);
```

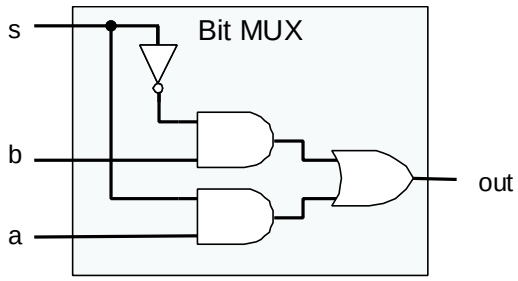


Figure 10: **Single-bit multiplexor circuit.** The output will equal input a if the control signal s is 1 and will equal input b when s is 0.

Our HCL expressions demonstrate a clear parallel between combinational logic circuits and logical expressions in C. They both use Boolean operations to compute functions over their inputs. Several differences between these two ways of expressing computation are worth noting:

- Since a combinational circuit consists of a series of logic gates, it has the property that the outputs continually respond to changes in the inputs. If some input to the circuit changes, then after some delay, the outputs will change accordingly. In contrast, a C expression is only evaluated when it is encountered during the execution of a program.
- Logical expressions in C allow arguments to be arbitrary integers, interpreting 0 as FALSE and anything else as TRUE. In contrast, our logic gates only operate over the bit values 0 and 1.
- Logical expressions in C have the property that they might only be partially evaluated. If the outcome of an AND or OR operation can be determined by just evaluating the first argument, then the second argument will not be evaluated. For example, with the C expression:

  ```
  (a && !a) && func(b,c)
  ```

  the function `func` will not be called, because the expression `(a && !a)` evaluates to 0. In contrast, combinational logic does not have any partial evaluation rules. The gates simply respond to changes on their inputs.

2.3 Word-Level Combinational Circuits and HCL Integer Expressions

By assembling large networks of logic gates, we can construct combinational circuits that compute much more complex functions. Typically, we design circuits that operate on data *words*. These are groups of bit-level signals that represent an integer or some control pattern. For example, our processor designs will contain numerous words, with word sizes ranging between 4 and 32 bits, representing integers, addresses, instruction codes, and register identifiers.

Combinational circuits to perform word-level computations are constructed using logic gates to compute the individual bits of the output word, based on the individual bits of the input word. For example, Figure 11 shows a combinational circuit that tests whether two 32-bit words A and B are equal. That is, the output

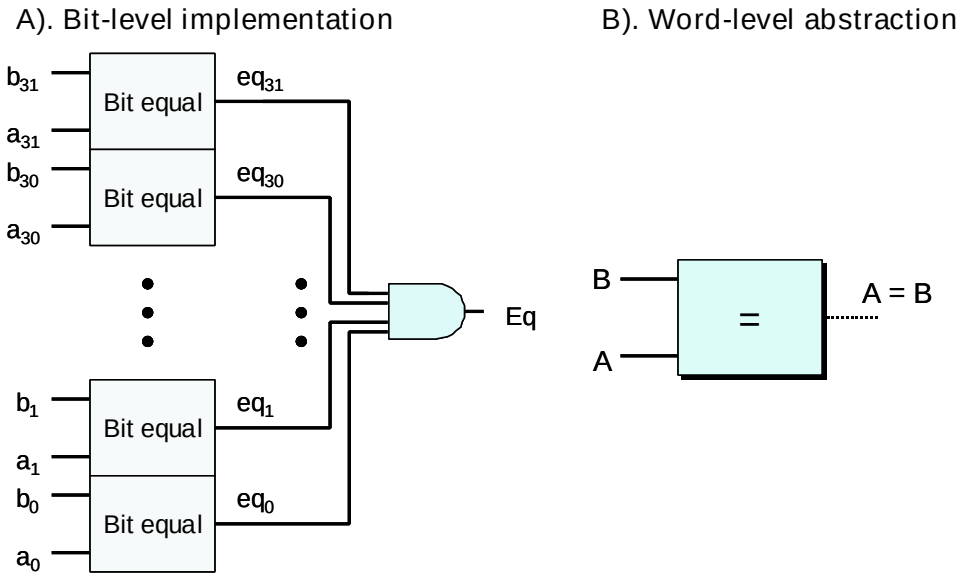


Figure 11: **Word-level equality test circuit.** The output will equal 1 when each bit from word A equals its counterpart from word B. Word-level equality is one of the operations in HCL.

will equal 1 if and only if each bit of A equals the corresponding bit of B. This circuit is implemented using 32 of the single-bit equality circuits shown in Figure 9. The outputs of these single-bit circuits are combined with an AND gate to form the circuit output.

In HCL, we will declare any word-level signal as an `int`, without specifying the word size. This is done for simplicity. In a full-featured hardware description language, every word can be declared to have a specific number of bits. HCL allows words to be compared for equality, and so the functionality of the circuit shown in Figure 11 can be expressed at the word level as

```
bool Eq = (A == B);
```

where arguments A and B are of type `int`. Note that we use the same syntax conventions as in C, where '=' denotes assignment, while '==' denotes the equality operator.

As is shown on the right side of Figure 11, we will draw word-level circuits using medium-thickness lines to represent the set of wires carrying the individual bits of the word, and we will show the resulting Boolean signal as a dashed line.

Practice Problem 7:

Suppose you want to implement a word-level equality circuit using the EXCLUSIVE-OR circuits from practice problem 6 rather than from bit-level equality circuits. Design such a circuit for a 32-bit word consisting of 32 bit-level EXCLUSIVE-OR circuits and two additional logic gates.

Figure 12 shows the circuit for a word-level multiplexor. This circuit generates a 32-bit word Out equal to one of the two input words, A or B, depending on the control input bit s. The circuit consists of 32 identical

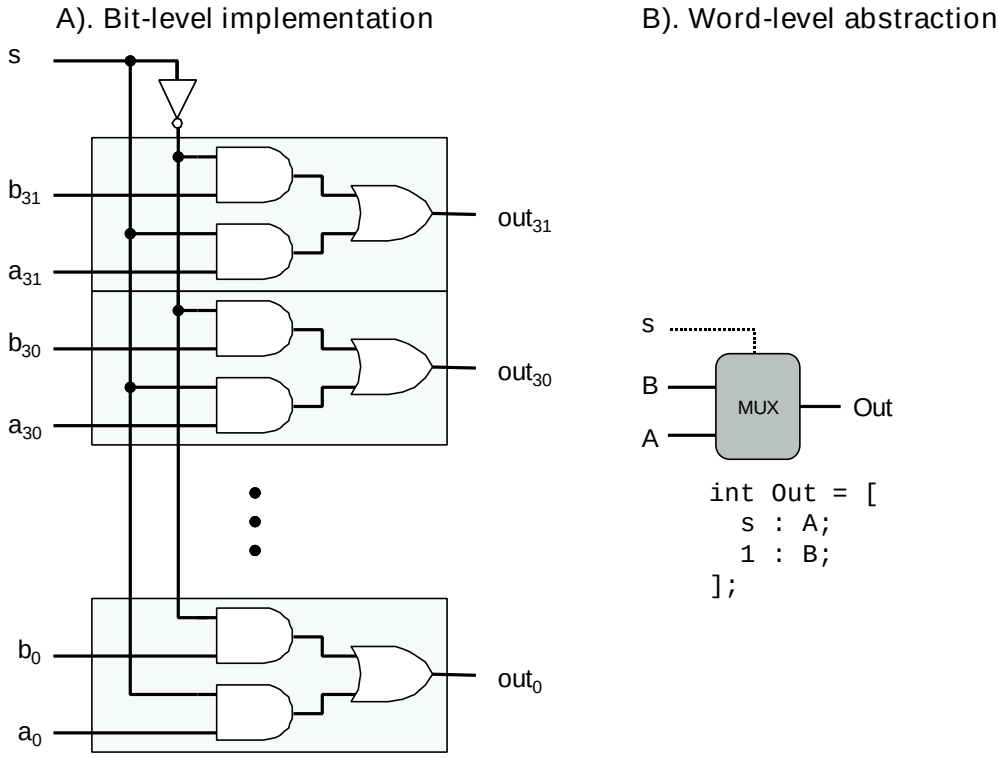


Figure 12: **Word-level multiplexor circuit.** The output will equal input word A when the control signal s is 1, and it will equal B otherwise. Multiplexors are described in HCL using case expressions.

subcircuits, each having a structure similar to the bit-level multiplexor from Figure 10. Rather than simply replicating the bit-level multiplexor 32 times, the word-level version reduces the number of inverters by generating `!s` once and reusing it at each bit position.

We will use many forms of multiplexors in our processor designs. They allow us to select a word from a number of sources depending on some control condition. Multiplexing functions are described in HCL using *case expressions*. A case expression has the following general form:

$$
\begin{array}{l}
\texttt{[} \\
\quad \begin{array}{lcl} select_1 & : & expr_1 \\ select_2 & : & expr_2 \\ & \vdots & \\ select_k & : & expr_k \end{array} \\
\texttt{]}
\end{array}
$$

The expression contains a series of cases, where each case i consists of a Boolean expression $select_i$, indicating when this case should be selected, and an integer expression $expr_i$, indicating the resulting value.

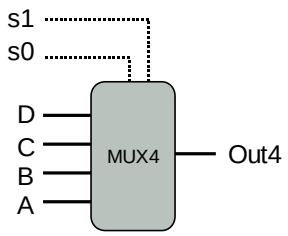
Unlike the switch statement of C, we do not require the different selection expressions to be mutually exclusive. Logically, the selection expressions are evaluated in sequence, and the case for the first one yielding 1 is selected. For example, the word-level multiplexor of Figure 12 can be described in HCL as:

```
int Out = [
        s: A;
        1: B;
];
```

In this code, the second selection expression is simply `1`, indicating that this case should be selected if no prior one has been. This is the way to specify a default case in HCL. Nearly all case expressions end in this manner.

Allowing nonexclusive selection expressions makes the HCL code more readable. An actual hardware multiplexor must have mutually exclusive signals controlling which input word should be passed to the output, such as the signals `s` and `!s` in Figure 12. To translate an HCL case expression into hardware, a logic synthesis program would need to analyze the set of selection expressions and resolve any possible conflicts by making sure that only the first matching case would be selected.

The selection expressions can be arbitrary Boolean expressions, and there can be an arbitrary number of cases. This allows case expressions to describe blocks where there are many choices of input signals with complex selection criteria. For example, consider the following diagram of a four-way multiplexor:

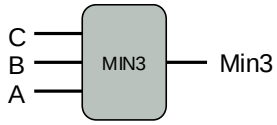


This circuit selects from among the four input words A, B, C, and D based on the control signals s1 and s0, treating the controls as a two-bit binary number. We can express this in HCL using Boolean expressions to describe the different combinations of control bit patterns:

```
int Out4 = [
        !s1 && !s0 : A; # 00
        !s1        : B; # 01
        s1 && !s0  : C; # 10
        1          : D; # 11
];
```

The comments on the right (any text starting with # and running for the rest of the line is a comment) show which combination of s1 and s0 will cause the case to be selected. Observe that the selection expressions can sometimes be simplified, since only the first matching case is selected. For example, the second expression can be written !s1, rather than the more complete !s1 && s0, since the only other possibility having s1 equal to 0 was given as the first selection expression.

As a final example, suppose we want design a logic circuit that finds the minimum value among a set of words A, B, and C, diagrammed as follows:



We can express this using an HCL case expression as follows:

```
int Min3 = [
        A <= B && A <= C : A;
        B <= A && B <= C : B;
        1                : C;
];
```

Practice Problem 8:

Write HCL code describing a circuit that for word inputs A, B, and C selects the *median* of the three values. That is, the output equals the word lying between the minimum and maximum of the three inputs.

Combinational logic circuits can be designed to perform many different types of operations on word-level data. The detailed design of these is beyond the scope of our presentation. One important combinational circuit, known as an *arithmetic/logic unit* (ALU), is diagrammed at an abstract level in Figure 13. This circuit has three inputs: two data inputs labeled A and B, and a control input. Depending on the setting of the control input, the circuit will perform different arithmetic or logical operations on the data inputs. Observe that the four operations diagrammed for this ALU correspond to the four different integer operations supported by the Y86 instruction set, and the control values match the function codes for these instructions (Figure 3). Note also the ordering of operands for subtraction, where the A input is subtracted from the B input. This ordering is chosen in anticipation of the ordering of arguments in the subl instruction.

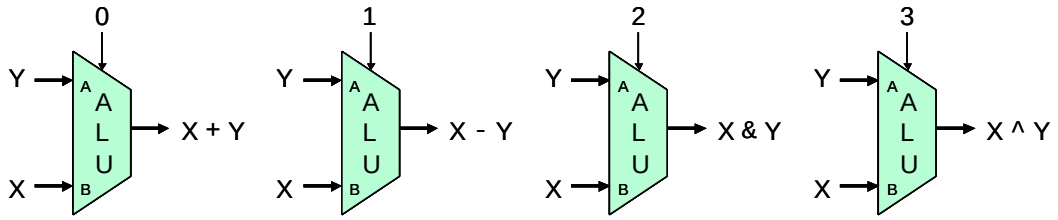
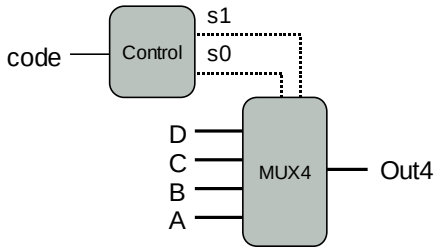


Figure 13: **Arithmetic/logic unit (ALU).** Depending on the setting of the function input, the circuit will perform one of four different arithmetic and logical operations.

2.4 Set Membership

In our processor designs, we will find many examples where we want to compare one signal against a number of possible matching signals, such as to test whether the code for some instruction being processed matches some category of instruction codes. As a simple example, suppose we want to generate the signals s1 and s0 for the four-way multiplexor of Figure 12 by selecting the high- and low-order bits from a two-bit signal code, as follows:



In this circuit, the two-bit signal code would then control the selection among the four data words A, B, C, and D. We can express the generation of signals s1 and s0 using equality tests based on the possible values of code:

```
bool s1 = code == 2 || code == 3;

bool s0 = code == 1 || code == 3;
```

A more concise expression can be written that expresses the property that s1 is 1 when code is in the set $\{2, 3\}$, and s0 is 1 when code is in the set $\{1, 3\}$:

```
bool s1 = code in { 2, 3 };

bool s0 = code in { 1, 3 };
```

The general form of a set membership test is

$$iexpr \quad \texttt{in} \quad \{iexpr_1, iexpr_2, \ldots, iexpr_k\}$$

where both the value being tested, $expr$, and the candidate matches, $iexpr_1$ through $iexpr_k$, are all integer expressions.

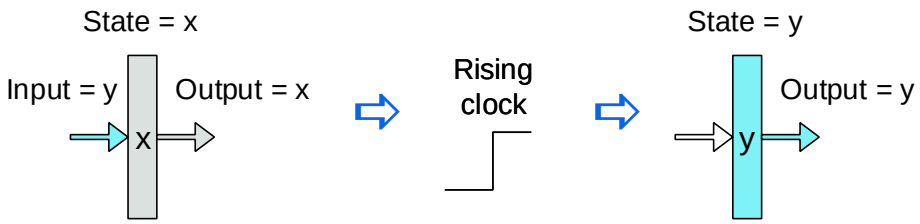


Figure 14: **Register operation.** The register outputs remain held at the current register state until the clock signal rises. When the clock rises, the values at the register inputs are captured to become the new register state.

2.5 Memory and Clocking

Combinational circuits, by their very nature, do not store any information. Instead, they simply react to the signals at their inputs, generating outputs equal to some function of the inputs. To create *sequential circuits*, that is, systems that have state and perform computations on that state, we must introduce devices that store information represented as bits. We consider two classes of memory devices:

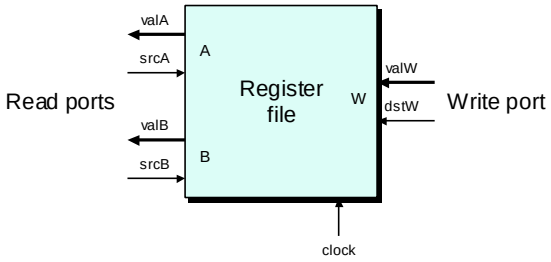
Clocked registers (or simply *registers*) store individual bits or words. A clock signal controls the loading of the register with the value at its input.

Random-access memories (or simply *memories*) store multiple words, using an address to select which word should be read or written. Examples of random-access memories include (1) the virtual memory system of a processor, where a combination of hardware and operating system software make it appear to a processor that it can access any word within a large address space; and (2) the register file, where register identifiers serve as the addresses. In an IA32 or Y86 processor, the register file holds the eight program registers (`%eax`, `%ecx`, etc.).

As we can see, the word "register" means two slightly different things when speaking of hardware versus machine-language programming. In hardware, a register is directly connected to the rest of the circuit by its input and output wires. In machine-level programming, the registers represent a small collection of addressable words in the CPU, where the addresses consist of register IDs. These words are generally stored in the register file, although we will see that the hardware can sometimes pass a word directly from one instruction to another to avoid the delay of first writing and then reading the register file. When necessary to avoid ambiguity, we will call the two classes of registers "hardware registers" and "program registers," respectively.

Figure 14 gives a more detailed view of a hardware register and how it operates. For most of the time, the register remains in a fixed state (shown as x), generating an output equal to its current state. Signals propagate through the combinational logic preceding the register, creating a new value for the register input (shown as y), but the register output remains fixed as long as the clock is low. As the clock rises, the input signals are loaded into the register as its next state (y), and this becomes the new register output until the next rising clock edge. A key point is that the registers serve as barriers between the combinational logic in different parts of the circuit. Values only propagate from a register input to its output once every clock cycle at the rising clock edge.

The following diagram shows a typical register file:



This register file has two *read ports*, named A and B, and one *write port*, named W. Such a *multiported* random-access memory allows multiple read and write operations to take place simultaneously. In the register file diagrammed, the circuit can read the values of two program registers and update the state of a third. Each port has an address input, indicating which program register should be selected, and a data output or input giving a value for that program register. The addresses are register identifiers, using the encoding shown in Figure 4. The two read ports have address inputs srcA and srcB (short for "source A" and "source B") and data outputs valA and valB (short for "value A" and "value B"). The write port has address input dstW (short for "destination W"), and data input valW (short for "value W").

Although the register file is not a combinational circuit (since it has internal storage), reading words from it operates in the same manner as a block of combinational logic having the addresses as inputs and the data as outputs. When either srcA or srcB is set to some register ID, then after some delay, the value stored in the corresponding program register will appear on either valA or valB. For example, setting srcA to 3 will cause the value of program register %ebx to be read, and this value will appear on output valA.

The writing of words to the register file is controlled by a clock signal in a manner similar to the loading of values into a clocked register. Every time the clock rises, the value on input valW is written to the program register indicated by the register ID on input dstW. When dstW is set to the special ID value 8, no program register is written.

3 Sequential Y86 Implementations

Now we have the components required to implement a Y86 processor. As a first step, we describe a processor called SEQ (for "sequential" processor). On each clock cycle, SEQ performs all the steps required to process a complete instruction. This would require a very long cycle time, however, and so the clock rate would be unacceptably low. Our purpose in developing SEQ is to provide a first step toward our ultimate goal of implementing an efficient, pipelined processor.

3.1 Organizing Processing into Stages

In general, processing an instruction involves a number of operations. We organize them in a particular sequence of stages, attempting to make all instructions follow a uniform sequence, even though the instructions differ greatly in their actions. The detailed processing at each step depends on the particular instruction being executed. Creating this framework will allow us to design a processor that makes best use

of the hardware. The following is an informal description of the stages and the operations performed within them:

Fetch: The fetch stage reads the bytes of an instruction from memory, using the program counter (PC) as the memory address. From the instruction it extracts the two four-bit portions of the instruction specifier byte, referred to as icode (the instruction code) and ifun (the instruction function). It possibly fetches a register specifier byte, giving one or both of the register operand specifiers rA and rB. It also possibly fetches a four-byte constant word valC. It computes valP to be the address of the instruction following the current one in sequential order. That is, valP equals the value of the PC plus the length of the fetched instruction.

Decode: The decode stage reads up to two operands from the register file, giving values valA and/or valB. Typically, it reads the registers designated by instruction fields rA and rB, but for some instructions it reads register `%esp`.

Execute: In the execute stage, the arithmetic/logic unit (ALU) either performs the operation specified by the instruction (according to the value of ifun), computes the effective address of a memory reference, or increments or decrements the stack pointer. We refer to the resulting value as valE. The condition codes are possibly set. For a jump instruction, the stage tests the condition codes and branch condition (given by ifun) to see whether or not the branch should be taken.

Memory: The memory stage may write data to memory, or it may read data from memory. We refer to the value read as valM.

Write back: The write-back stage writes up to two results to the register file.

PC update: The PC is set to the address of the next instruction.

The processor loops indefinitely, performing these stages. It stops only when it encounters a `halt` instruction or some error condition. The error conditions we handle are invalid memory addresses (either program or data) and invalid instructions.

As can be seen by the preceding description, there is a surprising amount of processing required to execute a single instruction. Not only must we perform the stated operation of the instruction, we must also compute addresses, update stack pointers, and determine the next instruction address. Fortunately, the overall flow can be similar for every instruction. Using a very simple and uniform structure is important when designing hardware, since we want to minimize the total amount of hardware, and we must ultimately map it onto the two-dimensional surface of an integrated circuit chip. One way to minimize the complexity is to have the different instructions share as much of the hardware as possible. For example, each of our processor designs contains a single arithmetic/logic unit that is used in different ways depending on the type of instruction being executed. The cost of duplicating blocks of logic in hardware is much higher than the cost of having multiple copies of code in software. It is also more difficult to deal with many special cases and idiosyncrasies in a hardware system than with software.

```
1   0x000: 308209000000 |     irmovl $9,  %edx
2   0x006: 308315000000 |     irmovl $21, %ebx
3   0x00c: 6123         |     subl %edx, %ebx        # subtract
4   0x00e: 308480000000 |     irmovl $128,%esp       # Practice Prob. 9
5   0x014: 404364000000 |     rmmovl %esp, 100(%ebx) # store
6   0x01a: a028         |     pushl %edx             # push
7   0x01c: b008         |     popl  %eax             # Practice Prob. 10
8   0x01e: 7328000000   |     je done                # Not taken
9   0x023: 8029000000   |     call proc              # Practice Prob. 13
10  0x028:              | done:
11  0x028: 10           |     halt
12  0x029:              | proc:
13  0x029: 90           |     ret                    # Return
```

Figure 15: **Sample Y86 instruction sequence.** We will trace the processing of these instructions through the different stages.

Stage	`OPl rA, rB`	`rrmovl rA, rB`	`irmovl V, rB`
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valP $\leftarrow$ PC + 2	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valP $\leftarrow$ PC + 2	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valC $\leftarrow$ M_4[PC + 2] valP $\leftarrow$ PC + 6
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[rB]	valA $\leftarrow$ R[rA]	
Execute	valE $\leftarrow$ valB OP valA Set CC	valE $\leftarrow$ 0 + valA	valE $\leftarrow$ 0 + valC
Memory			
Write back	R[rB] $\leftarrow$ valE	R[rB] $\leftarrow$ valE	R[rB] $\leftarrow$ valE
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP	PC $\leftarrow$ valP

Figure 16: **Computations in sequential implementation of Y86 instructions `OPl`, `rrmovl`, and `irmovl`.** These instructions compute a value and store the result in a register. The notation icode:ifun indicates the two components of the instruction byte, while rA:rB indicates the two components of the register specifier byte. The notation $M_1[x]$ indicates accessing (either reading or writing) one byte at memory location x, while $M_4[x]$ indicates accessing four bytes.

Our challenge is to arrange the computing required for each of the different instructions to fit within this general framework. We will use the code shown in Figure 15 to illustrate the processing of different Y86 instructions. Figures 16 through 19 contain tables describing how the different Y86 instructions proceed through the stages. It is worth the effort to study these tables carefully. They are in a form that enables a straightforward mapping into the hardware. Each line in these tables describes an assignment to some signal or stored state (indicated by the assignment operation $\leftarrow$). These should be read as if they were evaluated in sequence from top to bottom. When we later map the computations to hardware, we will find that we do not need to perform these evaluations in strict sequential order.

Figure 16 shows the processing required for instruction types `OPl` (integer and logical operations), `rrmovl` (register-register move), and `irmovl` (immediate-register move). Let's first consider the integer operations. Examining Figure 2, we can see that we have carefully chosen an encoding of instructions so that the four integer operations (`addl`, `subl`, `andl`, and `xorl`) all have the same value of icode. We can handle them all by an identical sequence of steps, except that the ALU computation must be set according to the particular instruction operation, encoded in ifun.

The processing of an integer-operation instruction follows the general pattern listed above. In the fetch stage, we do not require a constant word, and so valP is computed as PC + 2. During the decode stage, we read both operands. These are supplied to the ALU in the execute stage, along with the function specifier ifun, so that valE becomes the instruction result. This computation is shown as the expression valB OP valA, where OP indicates the operation specified by ifun. Note the ordering of the two arguments—this order is consistent with the conventions of Y86 (and IA32). For example, the instruction `subl %eax,%edx` is supposed to compute the value R[%edx] − R[%eax]. Nothing happens in the memory stage for these instructions, but valE is written to register rB in the write-back stage, and the PC is set to valP to complete the instruction execution.

Aside: Tracing the Execution of a `subl` Instruction.
As an example, let us follow the processing of the `subl` instruction on line 3 of the object code shown in Figure 15. We can see that the previous two instructions initialize registers `%edx` and `%ebx` to 9 and 21, respectively. We can also see that the instruction is located at address `0x00c` and consists of two bytes, having values `0x61` and `0x23`. The stages would proceed as shown in the following table, which lists the generic rule for processing an `OPl` instruction (Figure 16) on the left and the computations for this specific instruction on the right.

Stage	Generic	Specific
	`OPl rA, rB`	`subl %edx, %ebx`
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valP $\leftarrow$ PC + 2	icode:ifun $\leftarrow$ M_1[0x00c] = 6:1 rA:rB $\leftarrow$ M_1[0x00d] = 2:3 valP $\leftarrow$ 0x00c + 2 = 0x00e
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[rB]	valA $\leftarrow$ R[%edx] = 9 valB $\leftarrow$ R[%ebx] = 21
Execute	valE $\leftarrow$ valB OP valA Set CC	valE $\leftarrow$ 21 − 9 = 12 ZF $\leftarrow$ 0, SF $\leftarrow$ 0, OF $\leftarrow$ 0
Memory		
Write back	R[rB] $\leftarrow$ valE	R[%ebx] $\leftarrow$ valE = 12
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP = 0x00e

As this trace shows, we achieve the desired effect of setting register `%ebx` to 12, setting all three condition codes to 0, and incrementing the PC by 2. **End Aside.**

Stage	`rmmovl rA, D(rB)`	`mrmovl D(rB), rA`
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC+1] valC $\leftarrow$ M_4[PC + 2] valP $\leftarrow$ PC + 6	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC+1] valC $\leftarrow$ M_4[PC + 2] valP $\leftarrow$ PC + 6
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[rB]	valB $\leftarrow$ R[rB]
Execute	valE $\leftarrow$ valB + valC	valE $\leftarrow$ valB + valC
Memory	M_4[valE] $\leftarrow$ valA	valM $\leftarrow$ M_4[valE]
Write back		R[rA] $\leftarrow$ valM
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP

Figure 17: **Computations in sequential implementation of Y86 instructions `rmmovl` and `mrmovl`.** These instructions read or write memory.

Executing a `rrmovl` instruction proceeds much like an arithmetic operation. We do not need to fetch the second register operand, however. Instead, we set the second ALU input to 0 and add this to the first, giving valE = valA, which is then written to the register file. Similar processing occurs for `irmovl`, except that we use constant value valC for the first ALU input. In addition, we must increment the program counter by 6 for `irmovl` due to the long instruction format. Neither of these instructions changes the condition codes.

Practice Problem 9:

Fill in the right-hand column of the following table to describe the processing of the `irmovl` instruction on line 4 of the object code in Figure 15:

Stage	Generic	Specific
	`irmovl V, rB`	`irmovl $128, %esp`
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valC $\leftarrow$ M_4[PC + 2] valP $\leftarrow$ PC + 6	
Decode		
Execute	valE $\leftarrow$ 0 + valC	
Memory		
Write back	R[rB] $\leftarrow$ valE	
PC update	PC $\leftarrow$ valP	

How does this instruction execution modify the registers and the PC?

Stage	`pushl rA`	`popl rA`
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2
Decode	valA ← R[rA] valB ← R[%esp]	valA ← R[%esp] valB ← R[%esp]
Execute	valE ← valB + (−4)	valE ← valB + 4
Memory	M_4[valE] ← valA	valM ← M_4[valA]
Write back	R[%esp] ← valE	R[%esp] ← valE R[rA] ← valM
PC update	PC ← valP	PC ← valP

Figure 18: **Computations in sequential implementation of Y86 instructions `pushl` and `popl`.** These instructions push and pop the stack.

Figure 17 shows the processing required for the memory write and read instructions `rmmovl` and `mrmovl`. We see the same basic flow as before, but using the ALU to add valC to valB, giving the effective address (the sum of the displacement and the base register value) for the memory operation. In the memory stage we either write the register value valA to memory, or we read valM from memory.

Aside: Tracing the Execution of an `rmmovl` Instruction.
Let's trace the processing of the `rmmovl` instruction on line 5 of the object code shown in Figure 15. We can see that the previous instruction initialized register `%esp` to 128, while `%ebx` still holds 12, as computed by the `subl` instruction (line 3). We can also see that the instruction is located at address `0x014` and consists of six bytes. The first two have values `0x40` and `0x43`, while the final four are a byte-reversed version of the number `0x00000064` (decimal 100). The stages would proceed as follows:

Stage	Generic	Specific
	`rmmovl rA, D(rB)`	`rmmovl %esp, 100(%ebx)`
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC + 2] valP ← PC + 6	icode:ifun ← M_1[0x014] = 4:0 rA:rB ← M_1[0x015] = 4:3 valC ← M_4[0x016] = 100 valP ← 0x014 + 6 = 0x01a
Decode	valA ← R[rA] valB ← R[rB]	valA ← R[%esp] = 128 valB ← R[%ebx] = 12
Execute	valE ← valB + valC	valE ← 12 + 100 = 112
Memory	M_4[valE] ← valA	M_4[112] ← 128
Write back		
PC update	PC ← valP	PC ← 0x01a

As this trace shows, the instruction has the effect of writing 128 to memory address 112 and incrementing the PC by 6. **End Aside.**

Figure 18 shows the steps required to process `pushl` and `popl` instructions. These are among the most difficult Y86 instructions to implement, because they involve both accessing memory and incrementing

or decrementing the stack pointer. Although the two instructions have similar flows, they have important differences.

The `pushl` instruction starts much like our previous instructions, but in the decode stage we use `%esp` as the identifier for the second register operand, giving the stack pointer as value valB. In the execute stage, we use the ALU to decrement the stack pointer by 4. This decremented value is used for the memory write address and is also stored back to `%esp` in the write-back stage. By using valE as the address for the write operation, we adhere to the Y86 (and IA32) convention that `pushl` should decrement the stack pointer before writing, even though the actual updating of the stack pointer does not occur until after the memory operation has completed.

Aside: Tracing the Execution of a `pushl` Instruction.
Let's trace the processing of the `pushl` instruction on line 6 of the object code shown in Figure 15. At this point, we have 9 in register `%edx` and 128 in register `%esp`. We can also see that the instruction is located at address `0x01a` and consists of two bytes having values `0xa0` and `0x28`. The stages would proceed as follows:

Stage	Generic	Specific
	`pushl rA`	`pushl %edx`
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC+1] valP $\leftarrow$ PC + 2	icode:ifun $\leftarrow$ M_1[0x01a] = a:0 rA:rB $\leftarrow$ M_1[0x01b] = 2:8 valP $\leftarrow$ 0x01a + 2 = 0x01c
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[%esp]	valA $\leftarrow$ R[%edx] = 9 valB $\leftarrow$ R[%esp] = 128
Execute	valE $\leftarrow$ valB + (−4)	valE $\leftarrow$ 128 + (−4) = 124
Memory	M_4[valE] $\leftarrow$ valA	M_4[124] $\leftarrow$ 9
Write back	R[%esp] $\leftarrow$ valE	R[%esp] $\leftarrow$ 124
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ 0x01c

As this trace shows, the instruction has the effect of setting `%esp` to 124, writing 9 to address 124, and incrementing the PC by 2. **End Aside.**

The `popl` instruction proceeds much like `pushl`, except that we read two copies of the stack pointer in the decode stage. This is clearly redundant, but we will see that having the stack pointer as both valA and valB makes the subsequent flow more similar to that of other instructions, enhancing the overall uniformity of the design. We use the ALU to increment the stack pointer by 4 in the execute stage, but use the unincremented value as the address for the memory operation. In the write-back stage, we update both the stack pointer register with the incremented stack pointer, and register rA with the value read from memory. Using the unincremented stack pointer as the memory read address preserves the Y86 (and IA32) convention that `popl` should first read memory and then increment the stack pointer.

Practice Problem 10:

Fill in the right-hand column of the following table to describe the processing of the `popl` instruction on line 7 of the object code in Figure 15.

Stage	`jXX` Dest	`call` Dest	`ret`
Fetch	icode:ifun $\leftarrow$ M_1[PC] valC $\leftarrow$ M_4[PC + 1] valP $\leftarrow$ PC + 5	icode:ifun $\leftarrow$ M_1[PC] valC $\leftarrow$ M_4[PC + 1] valP $\leftarrow$ PC + 5	icode:ifun $\leftarrow$ M_1[PC] valP $\leftarrow$ PC + 1
Decode		 valB $\leftarrow$ R[%esp]	valA $\leftarrow$ R[%esp] valB $\leftarrow$ R[%esp]
Execute	 Bch $\leftarrow$ Cond(CC, ifun)	valE $\leftarrow$ valB + (−4)	valE $\leftarrow$ valB + 4
Memory		M_4[valE] $\leftarrow$ valP	valM $\leftarrow$ M_4[valA]
Write back		R[%esp] $\leftarrow$ valE	R[%esp] $\leftarrow$ valE
PC update	PC $\leftarrow$ Bch ? valC : valP	PC $\leftarrow$ valC	PC $\leftarrow$ valM

Figure 19: **Computations in sequential implementation of Y86 instructions `jXX`, `call`, and `ret`.** These instructions cause control transfers.

Stage	Generic	Specific
	`popl rA`	`popl %eax`
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC+1] valP $\leftarrow$ PC + 2	
Decode	valA $\leftarrow$ R[%esp] valB $\leftarrow$ R[%esp]	
Execute	valE $\leftarrow$ valB + 4	
Memory	valM $\leftarrow$ M_4[valA]	
Write back	R[%esp] $\leftarrow$ valE R[rB] $\leftarrow$ valM	
PC update	PC $\leftarrow$ valP	

What effect does this instruction execution have on the registers and the PC?

Practice Problem 11:

What would be the effect of the instruction `pushl %esp` according to the steps listed in Figure 18? Does this conform to the desired behavior for Y86, as determined in practice problem 4?

Practice Problem 12:

Assume the two register writes in the write-back stage for `popl` occur in the order listed in Figure 18. What would be the effect of executing `popl %esp`? Does this conform to the desired behavior for Y86, as determined in practice problem 5?

Figure 19 indicates the processing of our three control transfer instructions: the different jumps, `call`, and `ret`. We see that we can implement these instructions with the same overall flow as the preceding ones.

As with integer operations, we can process all of the jumps in a uniform manner, since they differ only when determining whether or not to take the branch. A jump instruction proceeds through fetch and decode much like the previous instructions, except that it does not require a register specifier byte. In the execute stage, we check the condition codes and the jump condition to determine whether or not to take the branch, yielding a 1-bit signal Bch. During the PC update stage, we test this flag, and set the PC to valC (the jump target) if the flag is 1, and to valP (the address of the following instruction) if the flag is 0. Our notation $x\ ?\ a : b$ is similar to the conditional expression in C—it yields a when x is nonzero and b when x is zero.

Aside: Tracing the Execution of a `je` Instruction.
Let's trace the processing of the `je` instruction on line 8 of the object code shown in Figure 15. The condition codes were all set to 0 by the `subl` instruction (line 3), and so the branch will not be taken. The instruction is located at address `0x01e` and consists of five bytes. The first has value `0x73`, while the remaining four are a byte-reversed version of the number `0x00000028`, the jump target. The stages would proceed as follows:

Stage	Generic	Specific
	`jXX` Dest	`je 0x028`
Fetch	icode:ifun $\leftarrow$ M_1[PC] valC $\leftarrow$ M_4[PC + 1] valP $\leftarrow$ PC + 5	icode:ifun $\leftarrow$ M_1[0x01e] = 7:3 valC $\leftarrow$ M_4[0x01f] = 0x028 valP $\leftarrow$ 0x01e + 5 = 0x023
Decode		
Execute	Bch $\leftarrow$ Cond(CC, ifun)	Bch $\leftarrow$ Cond($\langle 0, 0, 0 \rangle$, 3) = 0
Memory		
Write back		
PC update	PC $\leftarrow$ Bch ? valC : valP	PC $\leftarrow$ 0 ? 0x028 : 0x023 = 0x023

As this trace shows, the instruction has the effect of incrementing the PC by 5. **End Aside.**

Instructions `call` and `ret` bear some similarity to instructions `pushl` and `popl`, except that we push and pop program counter values. With instruction `call` we push valP, the address of the instruction that follows the `call` instruction. During the PC update stage, we set the PC to valC, the call destination. With instruction `ret`, we assign valM, the value popped from the stack, to the PC in the PC update stage.

Practice Problem 13:

Fill in the right-hand column of the following table to describe the processing of the `call` instruction on line 9 of the object code in Figure 15:

Stage	Generic	Specific
	`call` Dest	`call 0x029`
Fetch	icode:ifun $\leftarrow$ M_1[PC] valC $\leftarrow$ M_4[PC + 1] valP $\leftarrow$ PC + 5	
Decode	 valB $\leftarrow$ R[%esp]	
Execute	valE $\leftarrow$ valB + (−4)	
Memory	M_4[valE] $\leftarrow$ valP	
Write back	R[%esp] $\leftarrow$ valE	
PC update	PC $\leftarrow$ valC	

What effect would this instruction execution have on the registers, the PC, and the memory?

Aside: Tracing the Execution of an `ret` Instruction.
Let's trace the processing of the `ret` instruction on line 13 of the object code shown in Figure 15. The instruction address is `0x029` and is encoded by a single byte `0x90`. The previous `call` instruction set `%esp` to 124 and stored the return address `0x028` at memory address 124. The stages would proceed as follows:

Stage	Generic	Specific
	`ret`	`ret`
Fetch	icode:ifun $\leftarrow$ M_1[PC] valP $\leftarrow$ PC + 1	icode:ifun $\leftarrow$ M_1[0x029] = 9:0 valP $\leftarrow$ 0x029 + 1 = 0x02a
Decode	valA $\leftarrow$ R[%esp] valB $\leftarrow$ R[%esp]	valA $\leftarrow$ R[%esp] = 124 valB $\leftarrow$ R[%esp] = 124
Execute	valE $\leftarrow$ valB + 4	valE $\leftarrow$ 124 + 4 = 128
Memory	valM $\leftarrow$ M_4[valA]	valM $\leftarrow$ M_4[124] = 0x028
Write back	R[%esp] $\leftarrow$ valE	R[%esp] $\leftarrow$ 128
PC update	PC $\leftarrow$ valM	PC $\leftarrow$ 0x028

As this trace shows, the instruction has the effect of setting the PC to `0x028`, the address of the `halt` instruction. It also sets `%esp` to 128. **End Aside.**

We have created a uniform framework that handles all of the different types of Y86 instructions. Even though the instructions have widely varying behavior, we can organize the processing into six stages. Our task now is to create a hardware design that implements the stages and connects them together.

3.2 SEQ Hardware Structure

The computations required to implement all of the Y86 instructions can be organized as a series of six basic stages: fetch, decode, execute, memory, write back, and PC update. Figure 20 shows an abstract view of a

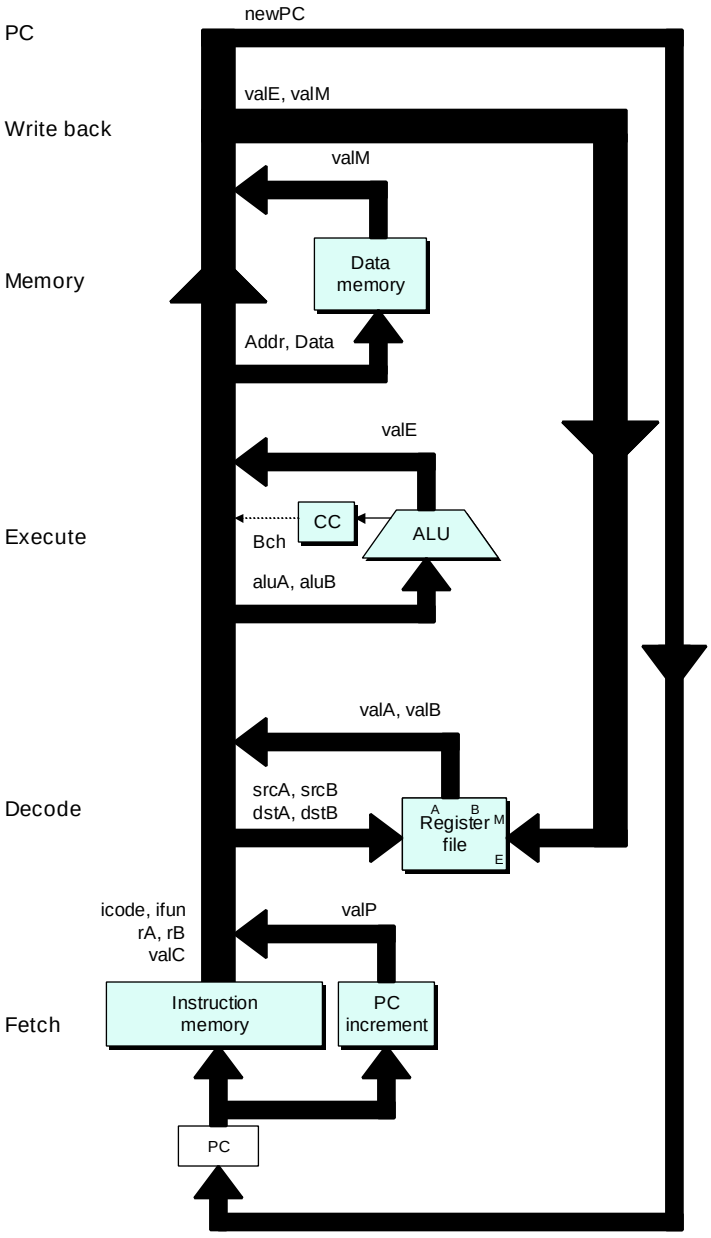


Figure 20: **Abstract view of SEQ, a sequential implementation.** The information processed during execution of an instruction follows a clockwise flow starting with an instruction fetch using the program counter (PC), shown in the lower left-hand corner of the figure.

hardware structure that can perform these computations. The program counter is stored in a register, shown in the lower left-hand corner (labeled "PC"). Information then flows along wires (shown grouped together as a heavy black line) first upward and then around to the right. Processing is performed by *hardware units* associated with the different stages. The feedback paths coming back down on the right-hand side contain the updated values to write to the register file and the updated program counter. This diagram omits some small blocks of combinational logic as well as all of the control logic needed to operate the different hardware units and to route the appropriate values to the units. We will add this detail later. Our method of drawing processors with the flow going from bottom to top is unconventional. We will explain the reason for our convention when we start designing pipelined processors.

The hardware units are associated with the different processing stages:

Fetch: Using the program counter register as an address, the instruction memory reads the bytes of an instruction. The PC incrementer computes valP, the incremented program counter.

Decode: The register file has two read ports, A and B via which register values valA and valB are read simultaneously.

Execute: The execute stage uses the arithmetic/logic (ALU) unit for different purposes according to the instruction type. For integer operations, it performs the specified operation. For other instructions it serves as an adder to compute an incremented or decremented stack pointer, to compute an effective address, or simply to pass one of its inputs to its outputs by adding 0.

The condition code register (CC) holds the three condition-code bits. New values for the condition codes are computed by the ALU. When executing a jump instruction, the branch signal Bch is computed based on the condition codes and the jump type.

Memory: The data memory reads or writes a word of memory when executing a memory instruction. The instruction and data memories access the same memory locations, but for different purposes.

Write back: The register file has two write ports. Port E is used to write values computed by the ALU, while port M is used to write values read from the data memory.

Figure 21 gives a more detailed view of the hardware required to implement SEQ (although we will not see the complete details until we examine the individual stages). We see the same set of hardware units as earlier, but now the wires are shown explicitly. In this figure, as well as in our other hardware diagrams, we use the following drawing conventions:

- *Hardware units are shown as light blue boxes.* These include the memories, the ALU, and so forth. We will use the same basic set of units for all of our processor implementations. We will treat these units as "black boxes" and not go into their detailed designs.
- *Control logic blocks are drawn as gray, rounded rectangles.* These blocks serve to select from among a set of signal sources, or to compute some Boolean function. We will examine these blocks in complete detail, including developing HCL descriptions.

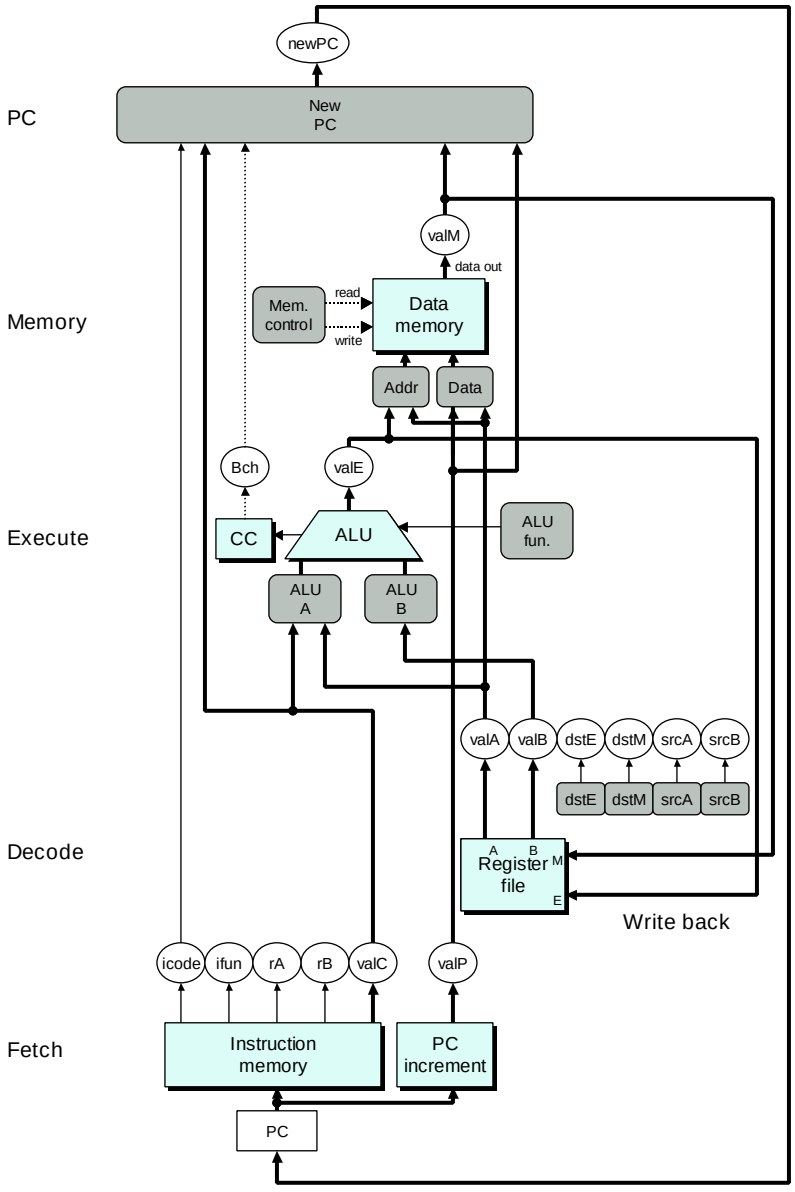


Figure 21: **Hardware structure of SEQ, a sequential implementation.** Some of the control signals, as well as the register and control word connections, are not shown.

Stage	Computation	OPl rA, rB	mrmovl D(rB), rA
Fetch	icode, ifun rA, rB valC valP	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valP $\leftarrow$ PC + 2	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC+1] valC $\leftarrow$ M_4[PC + 2] valP $\leftarrow$ PC + 6
Decode	valA, srcA valB, srcB	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[rB]	 valB $\leftarrow$ R[rB]
Execute	valE Cond. codes	valE $\leftarrow$ valB OP valA Set CC	valE $\leftarrow$ valB + valC
Memory	read/write		valM $\leftarrow$ M_4[valE]
Write back	E port, dstE M port, dstM	R[rB] $\leftarrow$ valE	 R[rA] $\leftarrow$ valM
PC update	PC	PC $\leftarrow$ valP	PC $\leftarrow$ valP

Figure 22: **Identifying the different computation steps in the sequential implementation.** The second column identifies the value being computed or the operation being performed in the stages of SEQ. The computations for instructions OPl and mrmovl are shown as examples of the computations.

- *Wires names are indicated in white, round boxes.* These are simply labels on the wires, not any kind of hardware element.
- *Word-wide data connections are shown as medium lines.* Each of these lines actually represents a bundle of 32 wires, connected in parallel, for transferring a word from one part of the hardware to another.
- *Byte and narrower data connections are shown as thin lines.* Each of these lines actually represents a bundle of 4 or 8 wires, depending on what type of values must be carried on the wires.
- *Single-bit connections are shown as dotted lines.* These represent control values passed between the units and blocks on the chip.

All of the computations we have shown in Figures 16 through 19 have the property that each line represents either the computation of a specific value, such as valP, or the activation of some hardware unit, such as the memory. These computations and actions are listed in the second column of Figure 22. In addition to the signals we have already described, this list includes four register ID signals: srcA, the source of valA; srcB, the source of valB; dstE, the register to which valE gets written; and dstM, the register to which valM gets written.

The two right-hand columns of this figure show the computations for the OPl and mrmovl instructions to illustrate the values being computed. To map the computations into hardware, we want to implement control logic that will transfer the data between the different hardware units and operate these units in such a way that the specified operations are performed for each of the different instruction types. That is the purpose of the control logic blocks, shown as gray, rounded boxes in Figure 21. Our task is to proceed through the individual stages and create detailed designs for these blocks.

3.3 SEQ Timing

In introducing the tables of Figures 16 through 19, we stated that they should be read as if they were written in a programming notation, with the assignments performed in sequence from top to bottom. On the other hand, the hardware structure of Figure 21, operates in fundamentally different way. Let's see how the hardware can implement the behavior listed in these tables.

Our implementation of SEQ consists of combinational logic and two forms of memory devices: clocked registers (the program counter and condition code register), and random-access memories (the register file, the instruction memory, and the data memory). Combinational logic does not require any sequencing or control—values propagate through a network of logic gates whenever the inputs change. As we mentioned, we can also view reading from a random-access memory as operating like combinational logic, with the output word generated based on the address input. Since our instruction memory is only used to read instructions, we can therefore treat this unit as if it were combinational logic.

We are left with just four hardware units that require an explicit control over their sequencing—the program counter, the condition code register, the data memory, and the register file. These are controlled via a single clock signal that triggers the loading of new values into the registers and the writing of values to the random-access memories. The program counter is loaded with a new instruction address every clock cycle. The condition code register is loaded only when an integer operation instruction is executed. The data memory is written only when an `rmmovl`, `pushl`, or `call` instruction is executed. The two write ports of the register file allow two program registers to be updated on every cycle, but we can use the special register ID 8 as a port address to indicate that no write should be performed for this port.

This clocking of the registers and memories is all that is required to control the sequencing of activities in our processor. Our hardware achieves the same effect as would a sequential execution of the assignments shown in the tables of Figures 16 through 19, even though all of the state updates actually occur simultaneously and only as the clock rises to start the next cycle. This equivalence holds because of the nature of the Y86 instruction set, and because we have organized the computations in such a way that our design obeys the following principle:

> The processor never needs to read back the state updated by an instruction in order to complete the processing of this instruction.

This principle is crucial to the success of our implementation.

As an illustration, suppose we implemented the `pushl` instruction by first decrementing `%esp` by 4 and then using the updated value of `%esp` as the address of a write operation. This approach would violate the principle stated above. It would require reading the updated stack pointer from the register file in order to perform the memory operation. Instead, our implementation (Figure 18) generates the decremented value of the stack pointer as the signal valE and then uses this signal both as the data for the register write and the address for the memory write. As a result, it can perform the register and memory writes simultaneously as the clock rises to begin the next clock cycle.

As another illustration of this principle, we can see that some instructions (the integer operations) set the condition codes, and some instructions (the jump instructions) read these condition codes, but no instruction must both set and then read the condition codes. Even though the condition codes are not set until the clock rises to begin the next clock cycle, they will be updated before any instruction attempts to read them.

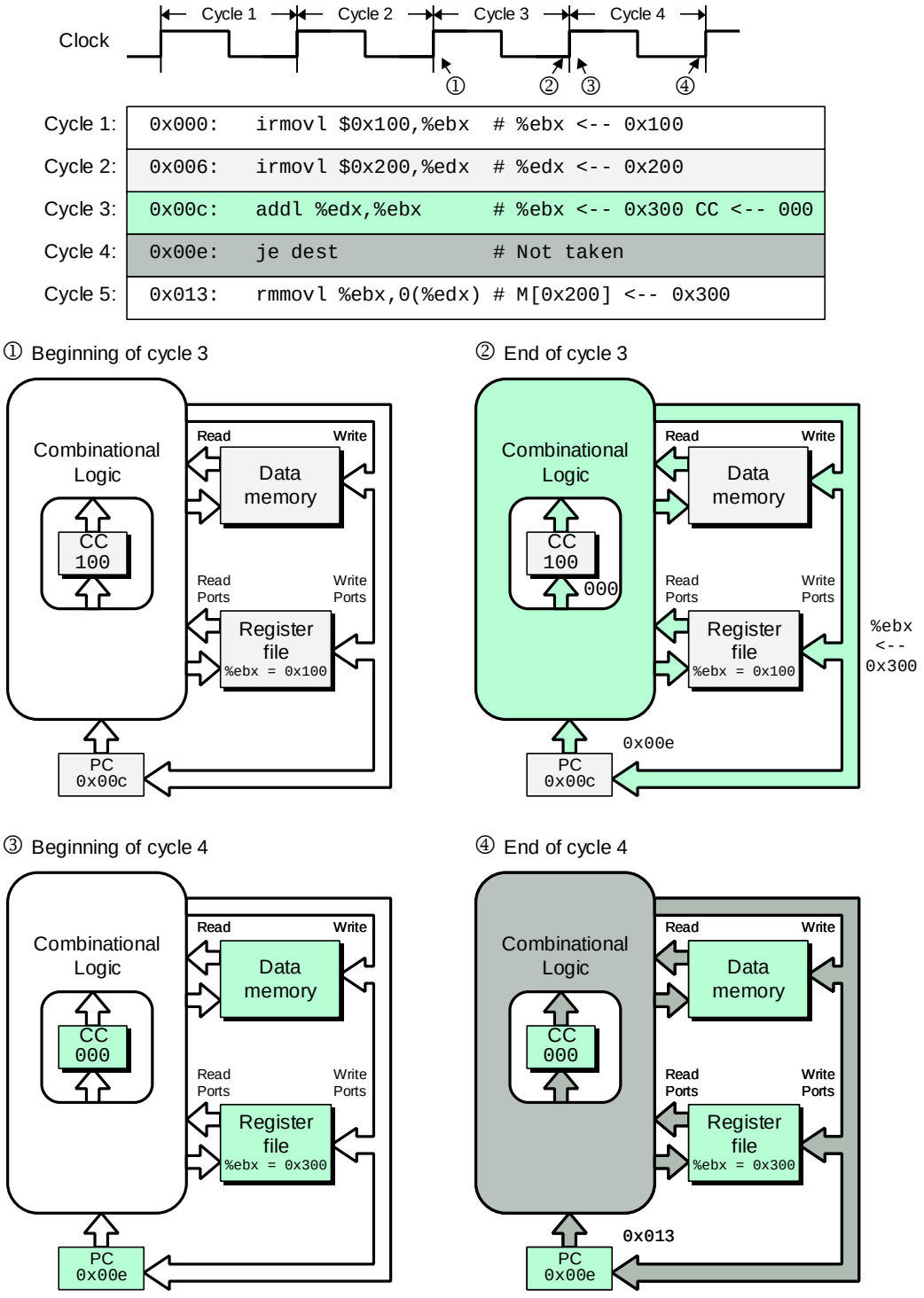


Figure 23: **Tracing two cycles of execution by SEQ.** Each cycle begins with the state elements (program counter, condition code register, register file, and data memory) set according to the previous instruction. Signals propagate through the combinational logic creating new values for the state elements. These values are loaded into the state elements to start the next cycle.

Figure 23 shows how the SEQ hardware would process the instructions at lines 3 and 4 in the following code sequence, shown in assembly code with the instruction addresses listed on the left:

```
1   0x000:   irmovl $0x100,%ebx  # %ebx <-- 0x100
2   0x006:   irmovl $0x200,%edx  # %edx <-- 0x200
3   0x00c:   addl %edx,%ebx      # %ebx <-- 0x300 CC <-- 000
4   0x00e:   je dest             # Not taken
5   0x013:   rmmovl %ebx,0(%edx) # M[0x200] <-- 0x300
6   0x019: dest: halt
```

Each of the diagrams labeled 1 through 4 shows the four state elements plus the combinational logic and the connections among the state elements. We show the combinational logic as being wrapped around the condition code register, because some of the combinational logic (such as the ALU) generates the input to the condition code register, while other parts (such as the branch computation and the PC selection logic) have the condition code register as input. We show the register file and the data memory as having separate connections for reading and writing, since the read operations propagate through these units as if they were combinational logic, while the write operations are controlled by the clock.

The color coding in Figure 23 indicates how the circuit signals relate to the different instructions being executed. We assume the processing starts with the condition codes, listed in the order `ZF`, `SF`, and `OF`, set to `100`. At the beginning of clock cycle 3 (point 1), the state elements hold the state as updated by the second `irmovl` instruction (line 2 of the listing), shown in light gray. The combinational logic is shown in white, indicating that it has not yet had time to react to the changed state. The clock cycle begins with address `0x00c` loaded into the program counter. This causes the `addl` instruction (line 3 of the listing), shown in blue, to be fetched and processed. Values flow through the combinational logic, including the reading of the random-access memories. By the end of the cycle (point 2), the combinational logic has generated new values (`000`) for the condition codes, an update for program register `%ebx`, and a new value (`0x00e`) for the program counter. At this point, the combinational logic has been updated according to the `addl` instruction (shown in blue), but the state still holds the values set by the second `irmovl` instruction (shown in light gray).

As the clock rises to begin cycle 4 (point 3), the updates to the program counter, the register file, and the condition code register occur, and so we show these in blue, but the combinational logic has not yet reacted to these changes, and so we show this in white. In this cycle, the `je` instruction (line 4 in the listing), shown in dark gray, is fetched and executed. Since condition code `ZF` is 0, the branch is not taken. By the end of the cycle (point 4), a new value of `0x00e` has been generated for the program counter. The combinational logic has been updated according to the `je` instruction (shown in dark gray), but the state still holds the values set by the `addl` instruction (shown in blue) until the next cycle begins.

As this example illustrates, the use of a clock to control the updating of the state elements, combined with the propagation of values through combinational logic, suffices to control the computations performed for each instruction in our implementation of SEQ. Every time the clock transitions from low to high, the processor begins executing a new instruction.

Name	Value (Hex)	Meaning
INOP	0	Code for nop instruction
IHALT	1	Code for halt instruction
IRRMOVL	2	Code for rrmovl instruction
IIRMOVL	3	Code for irmovl instruction
IRMMOVL	4	Code for rmmovl instruction
IMRMOVL	5	Code for mrmovl instruction
IOPL	6	Code for integer operation instructions
IJXX	7	Code for jump instructions
ICALL	8	Code for call instruction
IRET	9	Code for ret instruction
IPUSHL	a	Code for pushl instruction
IPOPL	b	Code for popl instruction
RESP	6	Register ID for %esp
RNONE	8	Indicates no register file access
ALUADD	0	Function for addition operation

Figure 24: **Constant values used in HCL descriptions.** These values represent the encodings of the instructions, register IDs, and ALU operations.

3.4 SEQ Stage Implementations

In this section, we devise HCL descriptions for the control logic blocks required to implement SEQ. A complete HCL description for SEQ is given in Section B of Appendix 6.1. We show some example blocks here, while others are given as practice problems. We recommend that you work these practice problems as a way to check your understanding of how the blocks relate to the computational requirements of the different instructions.

Part of the HCL description of SEQ that we do not include here is a definition of the different integer and Boolean signals that can be used as arguments to the HCL operations. These include the names of the different hardware signals, as well as constant values for the different instruction codes, register names, and ALU operations. The constants we use are documented in Figure 24. By convention, we use uppercase names for constant values.

In addition to the instructions shown in Figures 16 to 19, we include the processing for the nop and halt instructions. Both of these simply flow through stages without much processing, except to increment the PC by 1. We do not show the details of how the halt instruction actually stops the processor. We simply assume that the processor halts when it encounters a value of 1 for icode.

Fetch Stage

As shown in Figure 25, the fetch stage includes the instruction memory hardware unit. This unit reads six bytes from memory at a time, using the PC as the address of the first byte (byte 0). This byte is interpreted as the instruction byte and is split (by the unit labeled “Split”) into the two four-bit quantities icode and

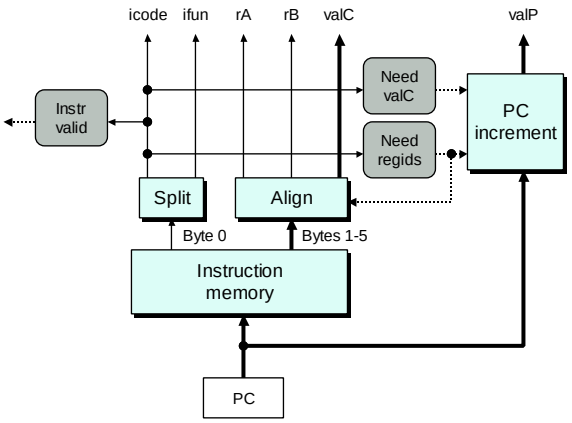


Figure 25: **SEQ fetch stage.** Six bytes are read from the instruction memory using the PC as the starting address. From these bytes we generate the different instruction fields. The PC increment block computes signal valP.

ifun. Based on the value of icode we can compute three one-bit signals (shown as dashed lines):

instr_valid: Does this byte correspond to a legal Y86 instruction? This signal is used to detect an illegal instruction.

need_regids: Does this instruction include a register specifier byte?

need_valC: Does this instruction include a constant word?

As an example, the HCL description for need_regids simply determines whether the value of icode is one of the instructions that has a register specifier byte.

```
bool need_regids =
        icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                     IIRMOVL, IRMMOVL, IMRMOVL };
```

Practice Problem 14:

Write HCL code for the signal `need_valC` in the SEQ implementation.

As Figure 25 shows, the remaining five bytes read from the instruction memory encode some combination of the register specifier byte and the constant word. These bytes are processed by the hardware unit labeled "Align" into the register fields and the constant word. When the computed signal need_regids is 1, then byte 1 is split into register specifiers rA and rB. Otherwise, these two fields are set to 8 (RNONE), indicating there are no registers specified by this instruction. Recall also (Figure 2) that for any instruction having only one register operand, the other field of the register specifier byte will be 8 (RNONE). Thus, we can assume that the signals rA and rB either encode registers we want to access or indicate that register access is not required. The unit labeled "Align" also generates the constant word valC. This will either be bytes 1 to 4 or bytes 2 to 5, depending on the value of signal need_regids.

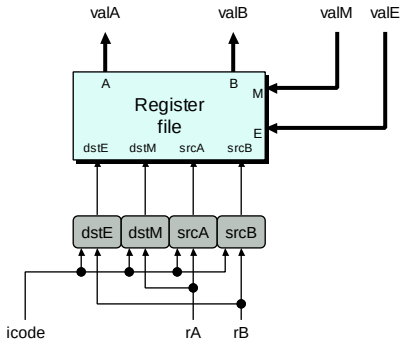


Figure 26: **SEQ decode and write-back stage.** The instruction fields are decoded to generate register identifiers for four addresses (two read and two write) used by the register file. The values read from the register file become the signals valA and valB. The two write-back values valE and valM serve as the data for the writes.

The PC incrementer hardware unit generates the signal valP, based on the current value of the PC, and the two signals need_regids and need_valC. For PC value p, need_regids value r, and need_valC value i, the incrementer generates the value $p + r + 4i$.

Decode and Write-Back Stages

Figure 26 provides a detailed view of logic that implements both the decode and write-back stages in SEQ. These two stages are combined because they both access the register file.

The register file has four ports. It supports up to two simultaneous reads (on ports A and B) and two simultaneous writes (on ports E and M). Each port has both an address connection and a data connection, where the address connection is a register ID, and the data connection is a set of 32 wires serving as either an output word (for a read port) or an input word (for a write port) of the register file. The two read ports have address inputs srcA and srcB, while the two write ports have address inputs dstA and dstB. The special identifier 8 (RNONE) on an address port indicates that no register should be accessed.

The four blocks at the bottom of Figure 26 generate the four different register IDs for the register file, based on the instruction code icode and the register specifiers rA and rB. Register ID srcA indicates which register should be read to generate valA. The desired value depends on the instruction type, as shown in the first row for the decode stage in Figures 16 to 19. Combining all of these entries into a single computation gives the following HCL description of srcA (recall that RESP is the register ID of %esp):

```
int srcA = [
        icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : rA;
        icode in { IPOPL, IRET } : RESP;
        1 : RNONE; # Don't need register
];
```

Practice Problem 15:

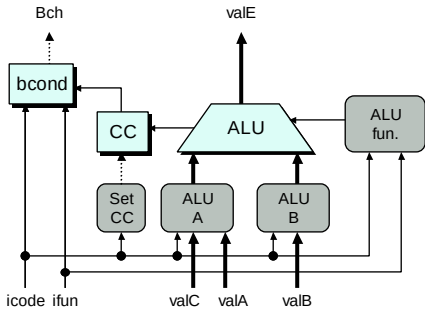


Figure 27: **SEQ execute stage.** The ALU either performs the operation for an integer operation instruction or it acts as an adder. The condition code registers are set according to the ALU value. The condition code values are tested to determine whether or not a branch should be taken.

The register signal srcB indicates which register should be read to generate the signal valB. The desired value is shown as the second step in the decode stage in Figures 16 to 19. Write HCL code for srcB.

Register ID dstE indicates the destination register for write port E, where the computed value valE is stored. This is shown in Figures 16 to 19 as the first step in the write-back stage. Combining the destination registers for all of the different instructions gives the following HCL description of dstE:

```
int dstE = [
        icode in { IRRMOVL, IIRMOVL, IOPL} : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't need register
];
```

Practice Problem 16:

Register ID dstM indicates the destination register for write port M, where valM, the value read from memory, is stored. This is shown in Figures 16 to 19 as the second step in the write-back stage. Write HCL code for dstM.

Practice Problem 17:

Only the `popl` instruction uses both of the register file write ports simultaneously. For the instruction `popl %esp`, the same address will be used for both the E and M write ports, but with different data. To handle this conflict, we must establish a *priority* among the two write ports so that when both attempt to write the same register on the same cycle, only the write from the higher priority port takes place. Which of the two ports should be given priority in order to implement the desired behavior, as determined in practice problem 5?

Execute Stage

The execute stage includes the arithmetic/logic unit (ALU.) This unit performs the operation ADD, SUBTRACT, AND, or EXCLUSIVE-OR, on inputs aluA and aluB based on the setting of the alufun signal. These

data and control signals are generated by three control blocks, as diagrammed in Figure 27. The ALU output becomes the signal valE.

In Figures 16 to 19, the ALU computation for each instruction is shown as the first step in the execute stage. The operands are listed with aluB first followed by aluA to make sure the `subl` instruction subtracts valA from valB. We can see that the value of aluA can be valA, valC, or either -4 or $+4$, depending on the instruction type. We can therefore express the behavior of the control block that generates aluA as follows:

```
int aluA = [
        icode in { IRRMOVL, IOPL } : valA;
        icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
        icode in { ICALL, IPUSHL } : -4;
        icode in { IRET, IPOPL } : 4;
        # Other instructions don't need ALU
];
```

Practice Problem 18:

Based on the first operand of the first step of the execute stage in Figures 16 to 19, write an HCL description for the signal aluB in SEQ.

Looking at the operations performed by the ALU in the execute stage, we can see that it is mostly used as an adder. For the `OPl` instructions, however, we want it to use the operation encoded in the ifun field of the instruction. We can therefore write the HCL description for the ALU control as follows:

```
int alufun = [
        icode == IOPL : ifun;
        1 : ALUADD;
];
```

The execute stage also includes the condition code register. Our ALU generates the three signals on which the condition codes are based—zero, sign, and overflow—every time it operates. However, we only want to set the condition codes when an `OPl` instruction is executed. We therefore generate a signal set_cc that controls whether or not the condition code register should be updated:

```
bool set_cc = icode in { IOPL };
```

The hardware unit labeled "bcond" determines whether or not an instruction should cause a jump (taken branch) or continue with the next instruction (not taken), generating control signal Bch. The instruction should cause a jump only if it is a jump instruction (icode equals `IJXX`) and the combination of condition code values and jump type (encoded in ifun) indicates a taken branch (see Figure **??**). We omit the detailed design of this unit.

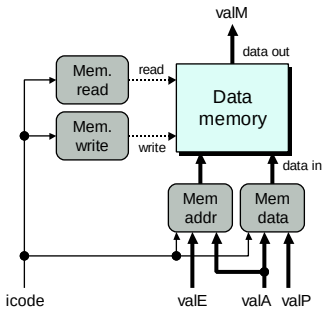


Figure 28: **SEQ memory stage.** The data memory can either write or read memory values. The value read from memory forms the signal valM.

Memory Stage

The memory stage has the task of either reading or writing program data. As shown in Figure 28, two control blocks generate the values for the memory address and the memory input data (for write operations). Two other blocks generate the control signals indicating whether to perform a read or a write operation. When a read operation is performed, the data memory generates the value valM.

The desired memory operation for each instruction type is shown in the memory stage of Figures 16 to 19. Observe that the address for memory reads and writes is always valE or valA. We can describe this block in HCL as follows:

```
int mem_addr = [
        icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
        icode in { IPOPL, IRET } : valA;
        # Other instructions don't need address
];
```

Practice Problem 19:

Looking at the memory operations for the different instructions shown in Figures 16 to 19, we can see that the data for memory writes is always either valA or valP. Write HCL code for the signal mem_data in SEQ.

We want to set the control signal mem_read only for instructions that read data from memory, as expressed by the following HCL code:

```
bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
```

Practice Problem 20:

We want to set the control signal mem_write only for instructions that write data to memory. Write HCL code for the signal mem_write in SEQ.

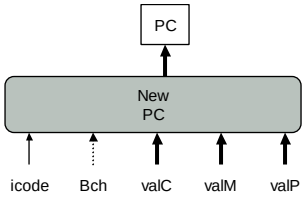


Figure 29: **SEQ PC update stage.** The next value of the PC is selected from among the signals valC, valM, and valP depending on the instruction code and the branch flag.

PC Update Stage

The final stage in SEQ generates the new value of the program counter. (See Figure 29). As the final steps in Figures 16 to 19 show, the new PC will be valC, valM, or valP, depending on the instruction type and whether or not a branch should be taken. This selection can be described in HCL as follows:

```
int new_pc = [
        # Call.  Use instruction constant
        icode == ICALL : valC;
        # Taken branch.  Use instruction constant
        icode == IJXX && Bch : valC;
        # Completion of RET instruction.  Use value from stack
        icode == IRET : valM;
        # Default: Use incremented PC
        1 : valP;
];
```

Surveying SEQ

We have now stepped through a complete design for a Y86 processor. We have seen that by organizing the steps required to execute each of the different instructions into a uniform flow, we can implement the entire processor with a small number of different hardware units and with a single clock to control the sequencing of computations. The control logic must then route the signals between these units and generate the proper control signals based on the instruction types and the branch conditions.

The only problem with SEQ is that it is too slow. The clock must run slowly enough so that signals can propagate through all of the stages within a single cycle. As an example, consider the processing of a `ret` instruction. Starting with an updated program counter at the beginning of the clock cycle, the instruction must be read from the instruction memory, the stack pointer must be read from the register file, the ALU must decrement the stack pointer, and the return address must be read from the memory in order to determine the next value for the program counter. All of this must be completed by the end of the clock cycle.

This style of implementation does not make very good use of our hardware units, since each unit is only active for a fraction of the total clock cycle. We will see that we can achieve much better performance by introducing pipelining.

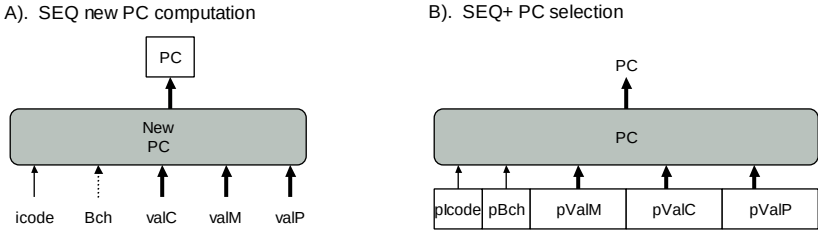
3.5 SEQ+: Rearranging the Computation Stages

As a transitional step toward a pipelined design, we rearrange the order of the six stages so that the PC stage comes at the beginning of the clock cycle, rather than at the end, yielding a processor design called SEQ+, because it extends the basic SEQ processor. This seems like a strange thing to do, because determining the new PC value can require testing the branch condition in the execute stage (for a conditional jump) or reading the return value in the memory stage (for `ret`).

As Figure 30 shows, we can move the PC stage so that its logic is active at the beginning of the clock cycle by making it compute the PC value for the *current* instruction. This PC value then feeds into the fetch stage, and the rest of the processing continues as before. The combinational logic generates all of the signals needed to compute the new PC value by the end of the clock cycle. These values are stored in a set of registers, shown in the figure as the box labeled “pState” (for “previous state”). The task of the PC stage now becomes to select the PC value for the current instruction rather than to compute an updated PC for the next instruction.

Figure 31 shows a more detailed view of the SEQ+ hardware. We can see that it contains the exact same hardware units and control blocks that we had in SEQ (Figure 21), but with the PC logic shifted to the bottom. The results from the previous instruction are stored in registers shown at the very bottom, labeled with the values they hold prefixed with the letter “p” (for “previous”).

The only change in the control logic is to redefine the PC computation so that it uses the previous state values. The following diagrams show the PC computation blocks for SEQ and SEQ+:



We see that the only difference between the two blocks is to shift the registers holding the processor state from after the PC computation to before. This an example of a general transformation known as *circuit retiming*. Retiming changes the state representation for a system without changing its logical behavior. It is often used to balance the delays between different sections of a system.

The HCL description of the PC computation becomes

```
int pc = [
        # Call.  Use instruction constant
        pIcode == ICALL : pValC;
        # Taken branch.  Use instruction constant
        pIcode == IJXX && pBch : pValC;
        # Completion of RET instruction.  Use value from stack
        pIcode == IRET : pValM;
        # Default: Use incremented PC
        1 : pValP;
];
```

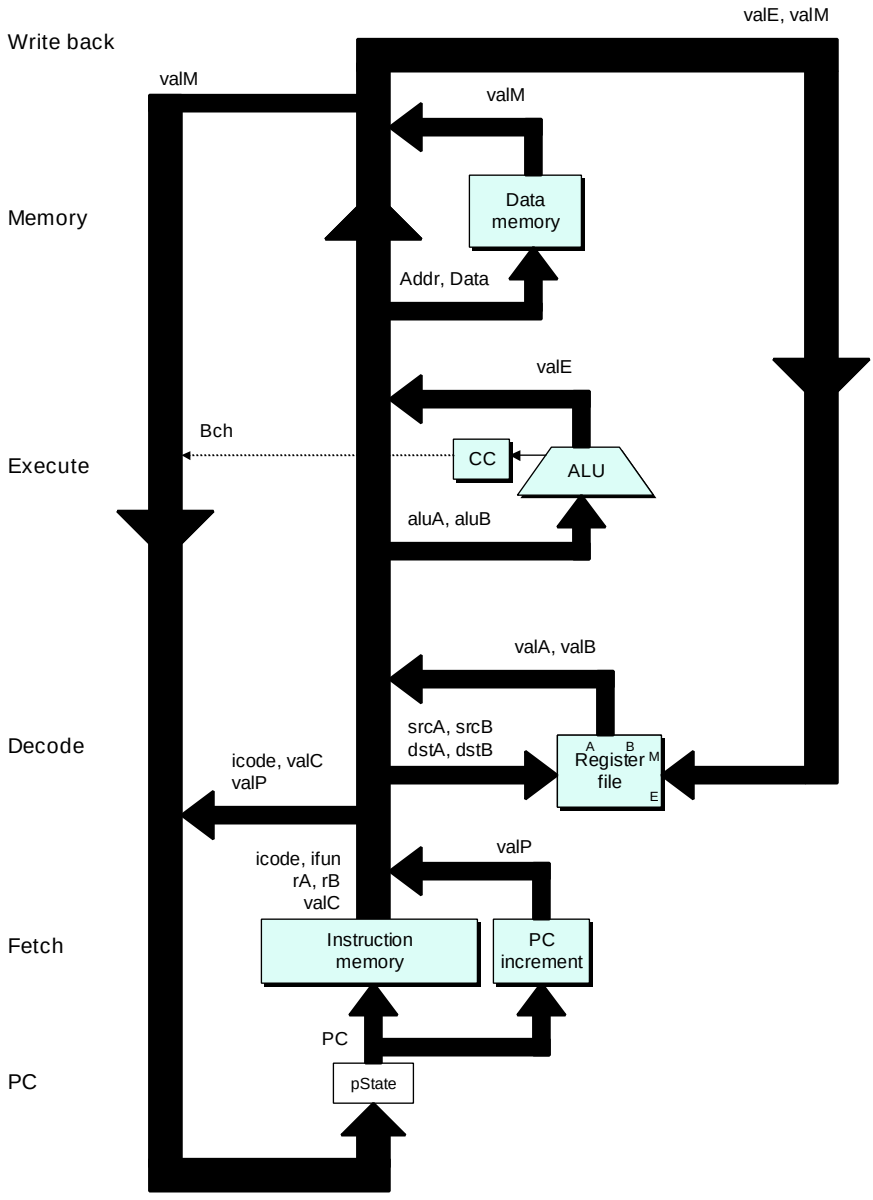


Figure 30: **SEQ+ abstract view.** In this version, the selection of the program counter (PC) for the current instruction is computed at the beginning of the cycle based on information (shown as “pState”) from the previous cycle. This structure will help us get to a pipelined implementation.

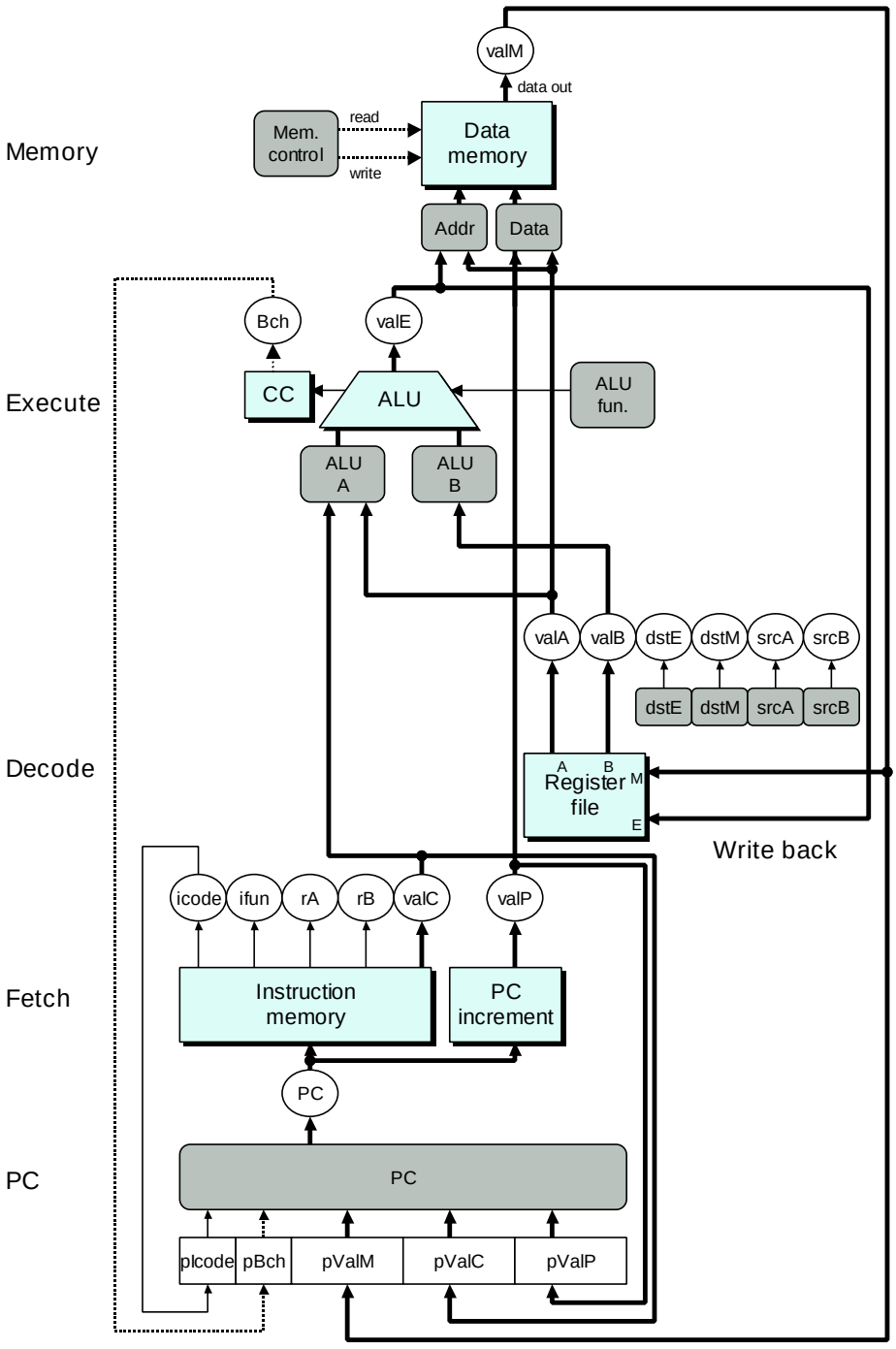


Figure 31: **SEQ+ hardware structure.** Some of the signals have been omitted.

The complete HCL description of SEQ+ is provided in Section C of Appendix 6.1.

Aside: Where's the PC in SEQ+?
One curious feature of SEQ+ is that there is no hardware register storing the program counter. Instead, the PC is computed dynamically based on some state information stored from the previous instruction. This is a small illustration of the fact that we can implement a processor in a way that differs from the conceptual model implied by the ISA, as long as the processor correctly executes arbitrary machine-language programs. We need not encode the state in the form indicated by the programmer-visible state, as long as the processor can generate correct values for any part of the programmer-visible state (such as the program counter). We will exploit this principle even more in creating a pipelined design. Out-of-order processing techniques, as described in Section **??** take this idea to an extreme by executing instructions in a completely different order than they occur in the machine-level program. **End Aside.**

4 General Principles of Pipelining

Before attempting to design a pipelined Y86 processor, let us consider some general properties and principles of pipelined systems. Such systems are familiar to anyone who has been through the serving line at a cafeteria or run a car through an automated car wash. In a pipelined system, the task to be performed is divided into a series of discrete stages. In a cafeteria, this involves supplying salad, a main dish, dessert, and beverage. In a car wash, this involves spraying water and soap, scrubbing, applying wax, and drying. Rather than having one customer run through the entire sequence from beginning to end before the next can begin, we allow multiple customers to proceed through the system at once. In a typical cafeteria line, the customers maintain the same order in the pipeline and pass through all stages, even if they do not want some of the courses. In the case of the car wash, a new car is allowed to enter the spraying stage as the preceding car moves from the spraying stage to the scrubbing stage. In general, the cars must move through the system at the same rate to avoid having one car crash into the next.

A key feature of pipelining is that it increases the *throughput* of the system, that is, the number of customers served per unit time, but it may also slightly increase the *latency*, that is, the time required to service an individual customer. For example, a customer in a cafeteria who only wants a salad could pass through a nonpipelined system very quickly, stopping only at the salad stage. A customer in a pipelined system who attempts to go directly to the salad stage risks incurring the wrath of other customers.

4.1 Computational Pipelines

Shifting our focus to computational pipelines, the "customers" are instructions and the stages perform some portion of the instruction execution. Figure 32 shows an example of a simple, nonpipelined hardware system. It consists of some logic that performs a computation, followed by a register to hold the results of this computation. A clock signal controls the loading of the register at some regular time interval. An example of such a system is the decoder in a compact disk (CD) player. The incoming signals are the bits read from the surface of the CD, and the logic decodes these to generate audio signals. The computational block in the figure is implemented as combinational logic, meaning that the signals will pass through a series of logic gates, with the outputs becoming some function of the inputs after some time delay.

In contemporary logic design, we measure circuit delays in units of *picoseconds* (abbreviated "ps"), or 10^{-12} seconds. In this example, we assume the combinational logic requires 300 picoseconds, while the

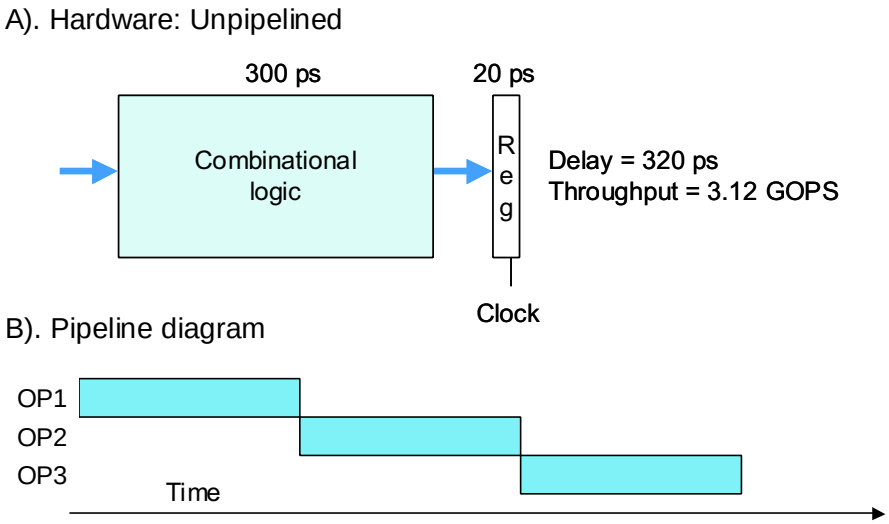


Figure 32: **Unpipelined computation hardware.** On each 320-ps cycle, the system spends 300 ps evaluating a combinational logic function and 20 ps storing the results in an output register.

loading of the register requires 20 ps. Figure 32 also shows a form of timing diagram known as a *pipeline diagram*. In this diagram, time flows from left to right. A series of operations (here named OP1, OP2, and OP3) are written from top to bottom. The solid rectangles indicate the times during which these operations are performed. In this system, we must complete one operation before beginning the next. Hence, the boxes do not overlap one another vertically. The following formula gives the maximum rate at which we could operate the system:

$$\text{Throughput} = \frac{1 \text{ operation}}{(20 + 300) \text{ picosecond}} \cdot \frac{1000 \text{ picosecond}}{1 \text{ nanosecond}} \approx 3.12 \text{ GOPS}$$

We express throughput in units of giga-operations per second (abbreviated GOPS), or billions of operations per second. The total time required to perform a single operation from beginning to end is known as the *latency*. In this system, the latency is 320 ps, the reciprocal of the throughput.

Suppose we could divide the computation performed by our system into three stages, A, B, and C, where each requires 100 ps, as illustrated in Figure 33. Then we could put *pipeline registers* between the stages so that each operation moves through the system in three steps, requiring three complete clock cycles from beginning to end. As the pipeline diagram in Figure 33 illustrates, we could allow OP2 to enter stage A as soon as OP1 moves from A to B, and so on. In steady state, all three stages would be active, with one operation leaving and a new one entering the system every clock cycle. We can see this during the third clock cycle in the pipeline diagram where OP1 is in stage C, OP2 is in stage B, and OP3 is in stage A. In this system, we could cycle the clocks every $100 + 20 = 120$ picoseconds, giving a throughput of around 8.33 GOPS. Since processing a single operation requires 3 clock cycles, the latency of this pipeline is $3 \times 120 = 360$ ps. We have increased the throughput of the system by a factor of $8.33/3.12 = 2.67$ at the expense of some added hardware and a slight increase in the latency ($360/320 = 1.12$). The increased latency is due to the time overhead of the added pipeline registers.

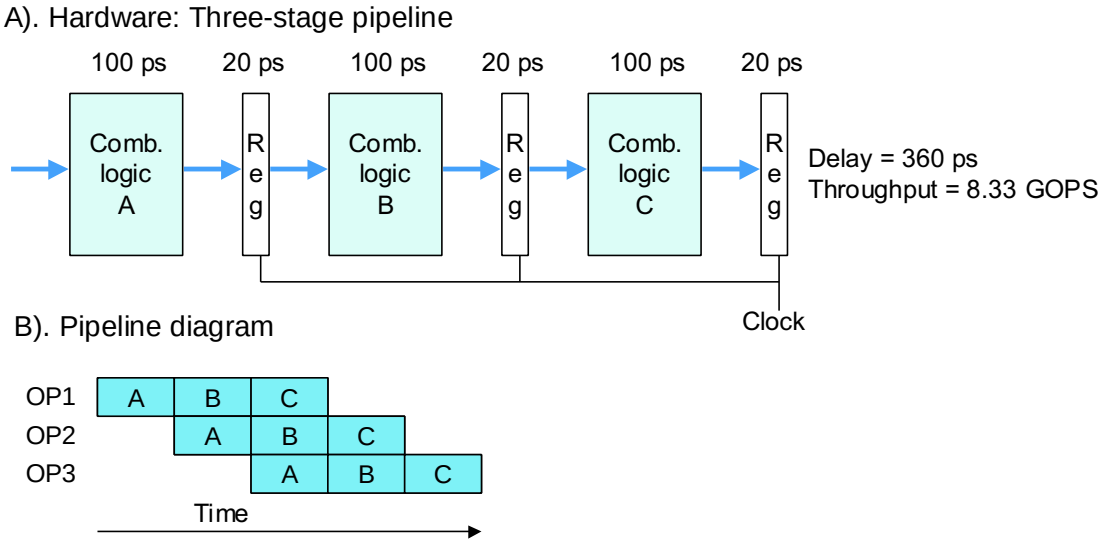


Figure 33: **Three-stage pipelined computation hardware.** The computation is split into stages A, B, and C. On each 120-ps cycle, each operation progresses through one stage.

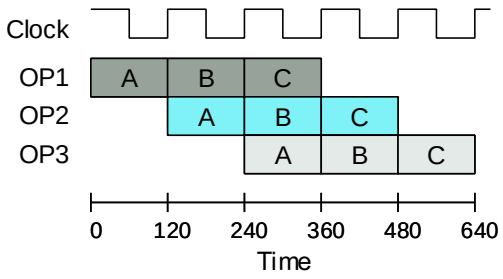


Figure 34: **Three-stage pipeline timing.** The rising edge of the clock signal controls the movement of operations from one pipeline stage to the next.

4.2 A Detailed Look at Pipeline Operation

To better understand how pipelining works, let's look in some detail at the timing and operation of pipeline computations. Figure 34 shows the pipeline diagram for the three-stage pipeline we've already looked at (Figure 33). The transfer of the operations between pipeline stages is controlled by a clock signal, as shown above the pipeline diagram. Every 120 ps, this signal rises from 0 to 1, initiating the next set of pipeline stage evaluations.

Figure 35 traces the circuit activity between times 240 and 360, as operation OP1 (shown in dark gray) propagates through stage C, OP2 (shown in blue) propagates through stage B, and OP3 (shown in light gray) propagates through stage A. Just before the rising clock at time 240 (point 1), the values computed in stage A for operation OP2 have reached the input of the first pipeline register, but its state and output remain set to those computed during stage A for operation OP1. The values computed in stage B for operation OP1 have reached the input of the second pipeline register. As the clock rises, these inputs are loaded

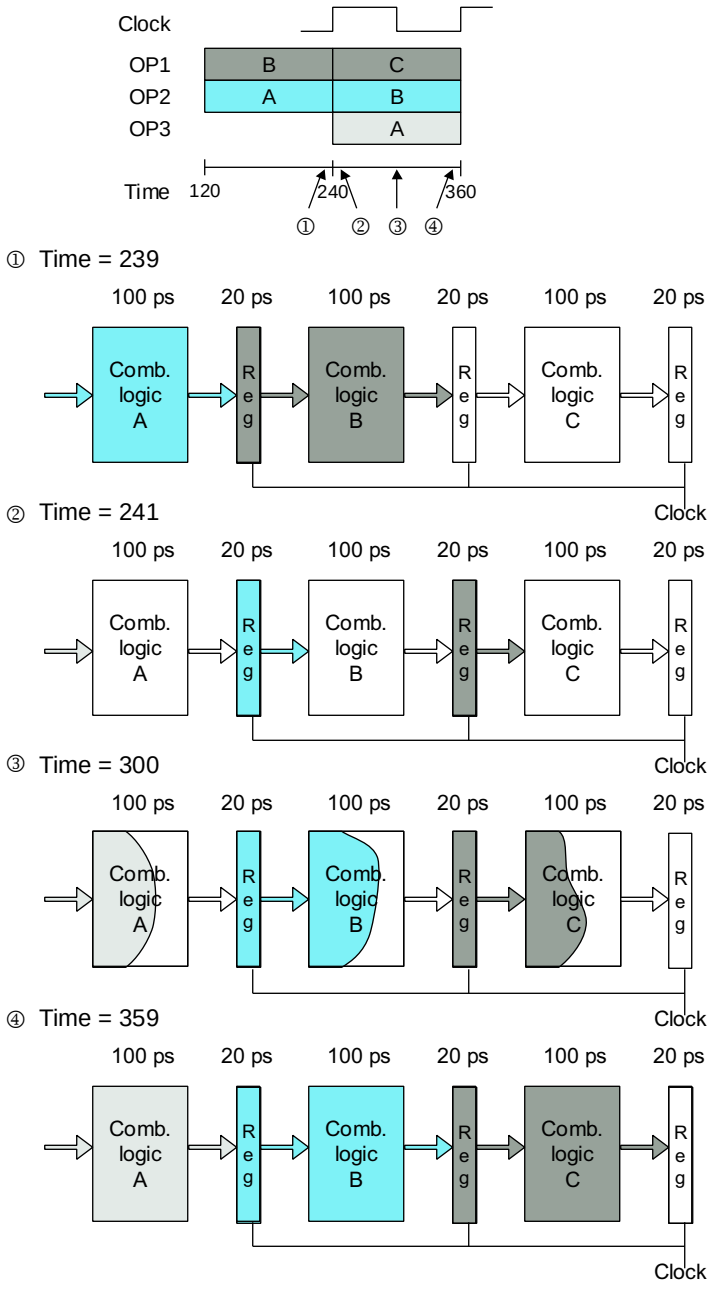


Figure 35: **One clock cycle of pipeline operation.** Just before the clock rises at time 240 (point 1), operations OP1 (shown in dark gray) and OP2 (shown in blue) have completed stages B and A. After the clock rises, these operations begin propagating through stages C and B, while operation OP3 (shown in light gray) begins propagating through stage A (points 2 and 3). Just before the clock rises again, the results for the operations have propagated to the inputs of the pipeline registers (point 4).

into the pipeline registers, becoming the register outputs (point 2). In addition, the input to stage A is set to initiate the computation of operation OP3. The signals then propagate through the combinational logic for the different stages (point 3). As the curved wavefronts in the diagram at point 3 suggest, signals can propagate through different sections at different rates. Before time 360, the result values reach the inputs of the pipeline registers (point 4). When the clock rises at time 360, each of the operations will have progressed through one pipeline stage.

We can see from this detailed view of pipeline operation that slowing down the clock operation would not change the pipeline behavior. The signals propagate to the pipeline register inputs, but no change in the register states will occur until the clock rises. On the other hand, we could have disastrous effects if the clock were run too fast. The values would not have time to propagate through the combinational logic, and so the register inputs would not yet be valid when the clock rises.

As with our discussion of the timing for the SEQ processor (Section 3.3), we see that the simple mechanism of having clocked registers between blocks of combinational logic suffices to control the flow of operations in the pipeline. As the clock rises and falls repeatedly, the different operations flow through the stages of the pipeline without interfering with one another.

4.3 Limitations of Pipelining

The example of Figure 33 shows an ideal pipelined system in which we are able to divide the computation into three independent stages, each requiring one-third of the time required by the original logic. Unfortunately, other factors often arise that diminish the effectiveness of pipelining.

Nonuniform Partitioning

Figure 36 shows a system in which we divide the computation into three stages as before, but the delays through the stages range from 50 to 150 ps. The sum of the delays through all of the stages remains 300 ps. However, the rate at which we can operate the clock is limited by the delay of the slowest stage. As the pipeline diagram in this figure shows, stage A will be idle (shown as a white box) for 100 ps every clock cycle, while stage C be idle for 50 ps every clock cycle. Only stage B will be continuously active. We must set the clock cycle to $150 + 20 = 170$ picoseconds, giving a throughput of 5.88 GOPS. In addition, the latency would increase to 510 ps due to the slower clock rate.

Devising a partitioning of the system computation into a series of stages having uniform delays can be a major challenge for hardware designers. Often, some of the hardware units in a processor, such as the ALU and the memories cannot be subdivided into multiple units with shorter delay. This makes it difficult to create a set of balanced stages. We will not concern ourselves with this level of detail in designing our pipelined Y86 processor, but it is important to appreciate the importance of timing optimization in actual system design.

Practice Problem 21:

Suppose we analyze the combinational logic of Figure 32 and determine that it can be separated into a sequence of six blocks, named A to F, having delays of 80, 30, 60, 50, 70, and 10 ps, respectively, illustrated as follows:

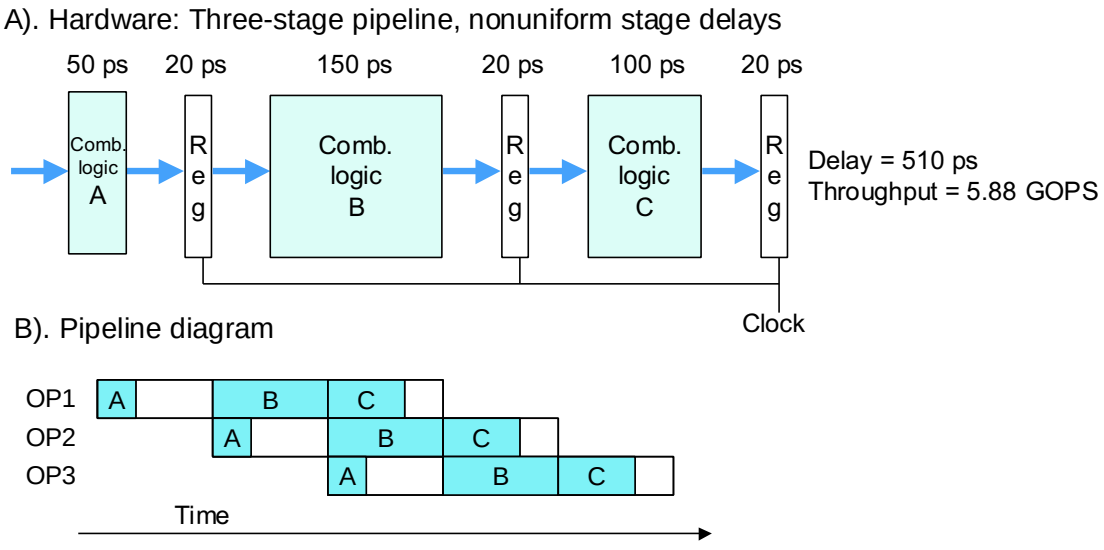
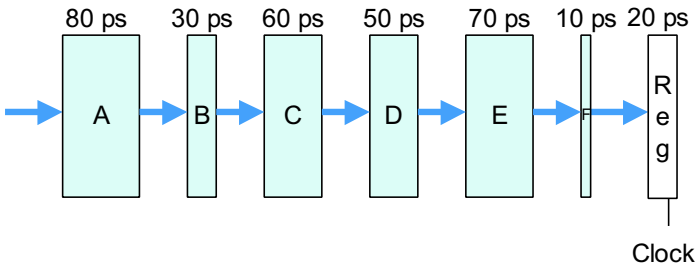


Figure 36: **Limitations of pipelining due to nonuniform stage delays.** The system throughput is limited by the speed of the slowest stage.



We can create pipelined versions of this design by inserting pipeline registers between pairs of these blocks. Different combinations of pipeline depth (how many stages) and maximum throughput arise, depending on where we insert the pipeline registers. Assume that a pipeline register has a delay of 20 ps.

A. Inserting a single register gives a two-stage pipeline. Where should the register be inserted to maximize throughput? What would be the throughput and latency?

B. Where should two registers be inserted to maximize the throughput of a three-stage pipeline? What would be the throughput and latency?

C. Where should three registers be inserted to maximize the throughput of a four-stage pipeline? What would be the throughput and latency?

D. What is the minimum number of stages that would yield a design with the maximum achievable throughput? Describe this design, its throughput, and its latency.

Diminishing Returns of Deep Pipelining

Figure 37 illustrates another limitation of pipelining. In this example we have divided the computation into six stages, each requiring 50 ps. Inserting a pipeline register between each pair of stages yields a six-stage

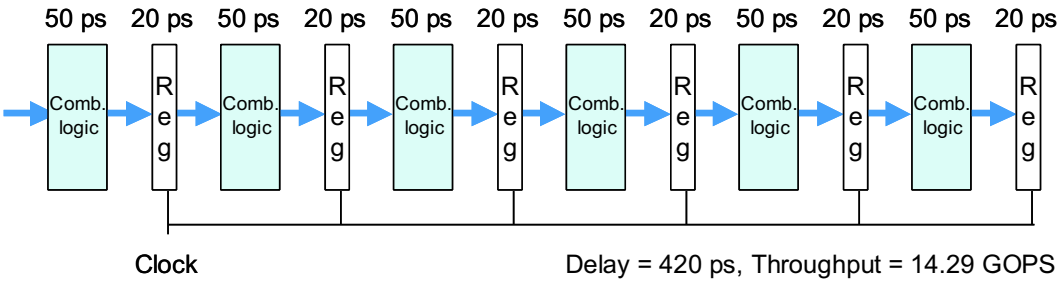


Figure 37: **Limitations of pipelining due to overhead.** As the combinational logic is split into shorter blocks, the delay due to register updating becomes a limiting factor.

pipeline. The minimum clock period for this system is $50 + 20 = 70$ picoseconds, giving a throughput of 14.29 GOPS. Thus, in doubling the number of pipeline stages, we improve the performance by a factor of $14.29/8.33 = 1.71$. Even though we have cut the time required for each computation block by a factor of two, we do not get a doubling of the throughput due to the delay through the pipeline registers. This delay becomes a limiting factor in the throughput of the pipeline. In our new design, this delay consumes 28.6% of the total total clock period.

Modern processors employ very deep (15 or more stages) pipelines in an attempt to maximize the processor clock rate. The processor architects divide the instruction execution in a large number of very simple steps so that each stage can have a very small delay. The circuit designers carefully design the pipeline registers to minimize their delay. The chip designers must also carefully design the clock distribution network to ensure that the clock changes at the exact same time across the entire chip. All of these factors contribute to the challenge of designing high-speed microprocessors.

Practice Problem 22:

Suppose we could take the system of Figure 32 and divide it into an arbitrary number of pipeline stages, all having the same delay. What would be the ultimate limit on the throughput, given pipeline register delays of 20 ps?

4.4 Pipelining a System with Feedback

Up to this point, we have considered only systems in which the objects passing through the pipeline—whether cars, people, or instructions—are completely independent of one another. For a system that executes machine programs such as IA32 or Y86, however, there are potential dependencies between successive instructions. For example, consider the following Y86 instruction sequence:

```
1    irmovl $50, %eax
2    addl %eax, %ebx
3    mrmovl 100( %ebx ), %edx
```

In this three-instruction sequence, there is a *data dependency* between each successive pair of instructions,

as indicated by the circled register names and the arrows between them. The `irmovl` instruction (line 1) stores its result in `%eax`, which then must be read by the `addl` instruction (line 2); and this instruction stores its result in `%ebx`, which must then be read by the `mrmovl` instruction (line 3).

Another source of sequential dependencies occurs due to the instruction control flow. Consider the following Y86 instruction sequence:

```
1 loop:
2     subl %edx,%ebx
3     jne targ
4     irmovl $10,%edx
5     jmp loop
6 targ:
7     halt
```

The `jne` instruction (line 3) creates a *control dependency* since the outcome of the conditional test determines whether the next instruction to execute will be the `irmovl` instruction (line 4) or the `halt` instruction (line 7). In our design for SEQ, these dependencies were handled by the feedback paths shown on the right-hand side of Figure 20. This feedback brings the updated register values down to the register file and the new PC value down to the PC register.

Figure 38 illustrates the perils of introducing pipelining into a system containing feedback paths. In the original system (A), the result of each operation is fed back around to the next operation. This is illustrated by the pipeline diagram (B), where the result of OP1 becomes an input to OP2, and so on. If we attempt to convert this to a three-stage pipeline (C), we change the behavior of the system. As the pipeline diagram (C) shows, the result of OP1 becomes an input to OP4. In attempting to speed up the system via pipelining, we have changed the system behavior.

When we introduce pipelining into a Y86 processor, we must deal with feedback effects properly. Clearly, it would be unacceptable to alter the system behavior as occurred in the example of Figure 38. Somehow we must deal with the data and control dependencies between instructions so that the resulting behavior matches the model defined by the ISA.

5 Pipelined Y86 Implementations

We are finally ready for the major task of this chapter—designing a pipelined Y86 processor. We start with SEQ+ as our basis and add pipeline registers between the stages. Our first attempt at this does not handle the different data and control dependencies properly. By making some modifications, however, we achieve our goal of an efficient pipelined processor that implements the Y86 ISA.

5.1 Inserting Pipeline Registers

In our first attempt at creating a pipelined Y86 processor, we insert pipeline registers between the stages of SEQ+ and rearrange signals somewhat, yielding the PIPE– processor, where the "–" in the name signifies that this processor has somewhat less performance than our ultimate processor design. The abstract structure

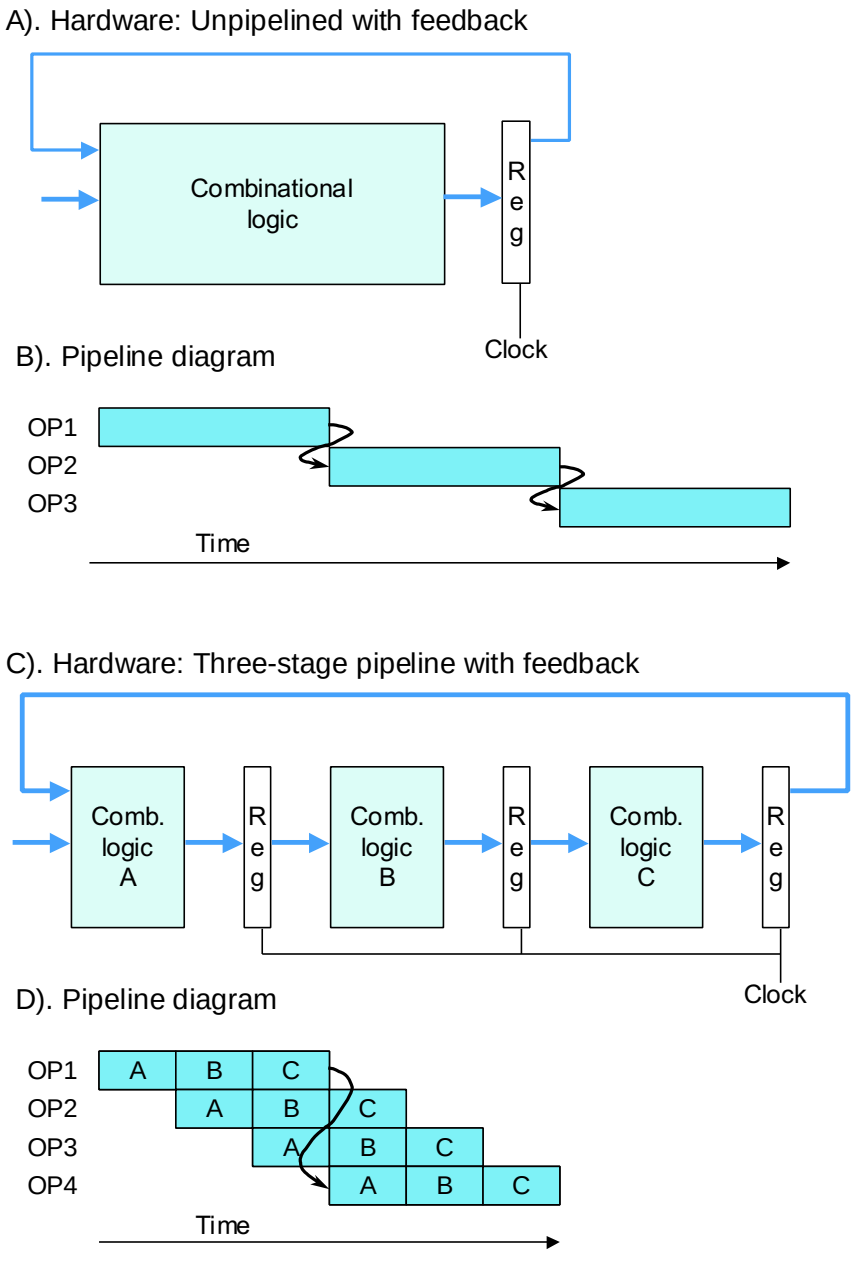


Figure 38: **Limitations of pipelining due to logical dependencies.** In going from an unpipelined system with feedback (A) to a pipelined one (C), we change its computational behavior, as can be seen by the two pipeline diagrams (B and D).

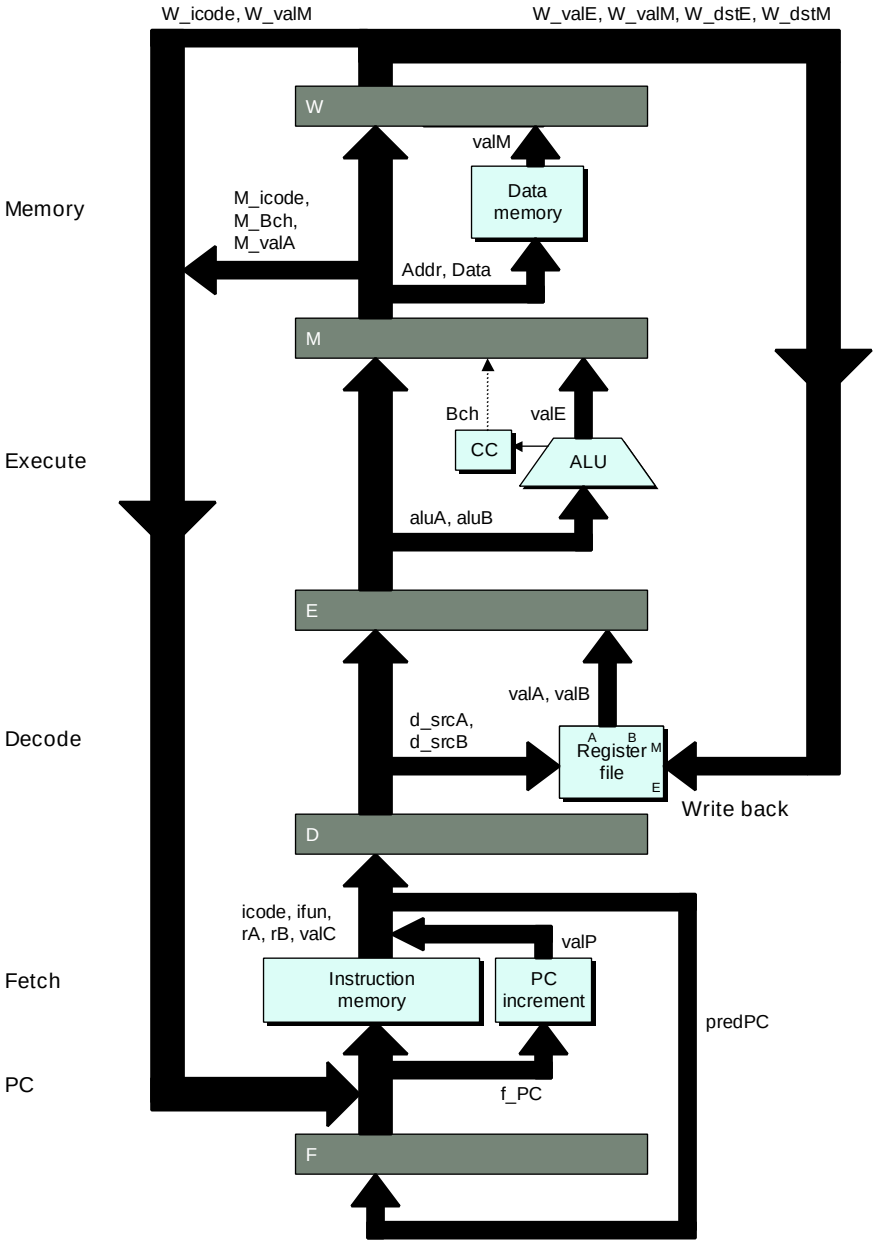


Figure 39: **Abstract view of PIPE–, an initial pipelined implementation.** By inserting pipeline registers into SEQ+ (Figure 30), we create a five-stage pipeline. There are several shortcomings of this version that we will deal with shortly.

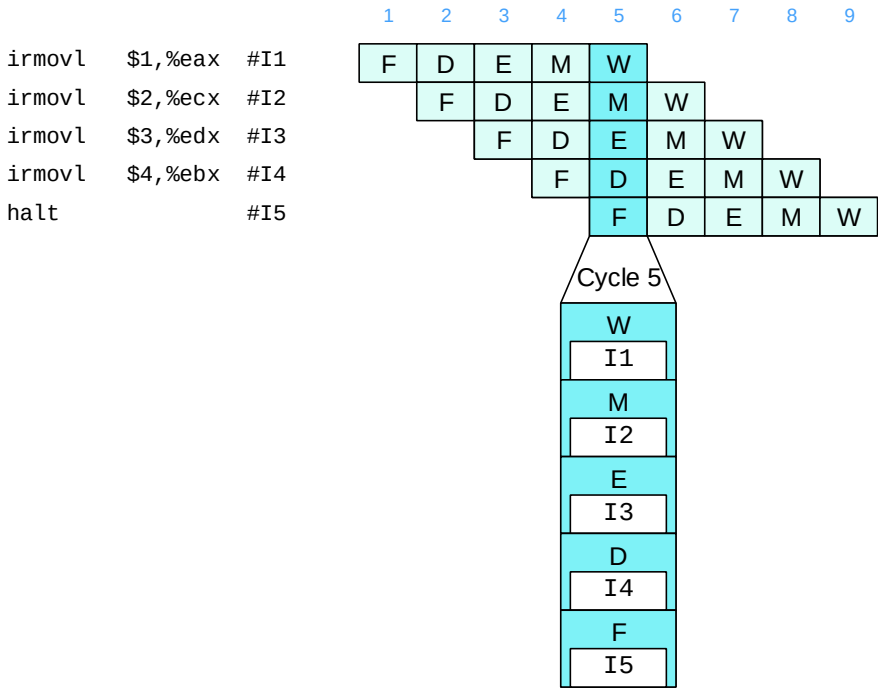


Figure 40: **Example of instruction flow through pipeline.**

of PIPE– is illustrated in Figure 39. The pipeline registers are shown in this figure as gray boxes. Each of these registers holds multiple bytes and words, as we will examine later. Observe that PIPE– uses the exact same set of hardware units as our two sequential designs: SEQ (Figure 20) and SEQ+ (Figure 30).

The pipeline registers are labeled as follows:

F holds a *predicted* value of the program counter, as will be discussed shortly.

D sits between the fetch and decode stages. It holds information about the most recently fetched instruction for processing by the decode stage.

E sits between the decode and execute stages. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage.

M sits between the execute and memory stages. It holds the results of the most recently executed instruction for processing by the memory stage. It also holds information about branch conditions and branch targets for processing conditional jumps.

W sits between the memory stage and the feedback paths that supply the computed results to the register file for writing and the return address to the PC selection logic when completing a `ret` instruction.

Figure 40 shows how the following code sequence would flow through our five-stage pipeline, where the comments identify the instructions as `I1` to `I5` for reference:

```
1   irmovl  $1,%eax  # I1
2   irmovl  $2,%ecx  # I2
3   irmovl  $3,%edx  # I3
4   irmovl  $4,%ebx  # I4
5   halt             # I5
```

The right side of the figure shows a pipeline diagram for this instruction sequence. As with the pipeline diagrams for the simple pipelined computation units of Section 4, this diagram show the progression of each instruction through the pipeline stages, with time increasing from left to right. The numbers along the top identify the clock cycles at which the different stages occur. For example, in cycle 1, instruction `I1` is fetched, and it then proceeds through the pipeline stages, with its result being written to the register file after the end of cycle 5. Instruction I2 is fetched in cycle 2, and its result is written back after the end of cycle 6, and so on. At the bottom, we show an expanded view of the pipeline for cycle 5. At this point, there is an instruction in each of the pipeline stages.

From Figure 40, we can also justify our convention of drawing processors so that the instructions flow from bottom to top. The expanded view for cycle 5 shows the pipeline stages with the fetch stage on the bottom and the write-back stage on the top, just as do our diagrams of the pipeline hardware (Figures 39 and 41). If we look at the ordering of instructions in the pipeline stages, we see that they appear in the same order as they do in the program listing. Since normal program flow goes from top to bottom of a listing, we preserve this ordering by having the pipeline flow go from bottom to top. This convention is particularly useful when working with the simulators that accompany this text.

Figure 41 gives a more detailed view of the PIPE– hardware structure. We can see that each pipeline register contains multiple fields (shown as white boxes), corresponding to the signals associated with the different instructions flowing through the pipeline. Unlike the labels shown in rounded boxes in the hardware structure of the two sequential processors (Figures 21 and 31), these white boxes represent actual hardware components.

Comparing the abstract structure of SEQ+ (Figure 30) to that of PIPE– (Figure 39), we see that although the overall flows through the stages are very similar, there are some subtle differences. We examine these differences before proceeding with a detailed implementation.

5.2 Rearranging and Relabeling Signals

SEQ+ only processes one instruction at a time, and so there are unique values for signals such as valC, srcA, and valE. In our pipelined design, there will be multiple versions of these values associated with the different instructions flowing through the system. For example, in the detailed structure of PIPE–, there are four white boxes labeled "icode" that hold the icode signals for four different instructions. (See Figure 41.) We need to take great care to make sure we use the proper version of a signal, or else we could have serious errors, such as storing the result computed for one instruction at the destination register specified by another instruction. We adopt a naming scheme where a signal stored in a pipeline register can be uniquely identified by prefixing its name with that of the pipe register written in upper case. For example, the four copies of icode are named D_icode, E_icode, M_icode, and W_icode. We also need to refer to some signals that have just been computed within a stage. These are labeled by prefixing the signal name with the first character of the stage name, written in lower case. Examples include d_srcA and e_Bch.

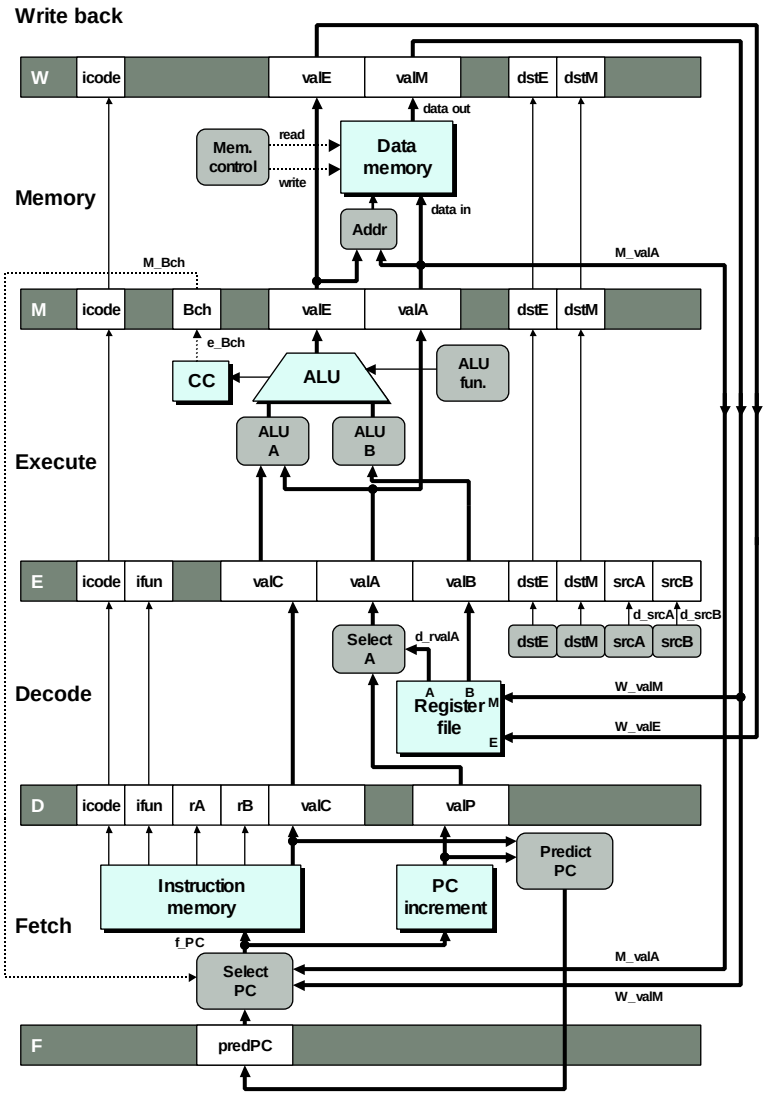


Figure 41: **Hardware structure of PIPE–, an initial pipelined implementation.** Not all connections are shown.

The decode stages of SEQ+ and PIPE– both generate signals dstE and dstM indicating the destination register for values valE and valM. In SEQ+, we could connect these signals directly to the address inputs of the register file write ports. With PIPE–, these signals are carried along in the pipeline through the execute and memory stages, and are directed to the register file only once they reach the write-back stage (shown in the more detailed views of the stages). We do this to make sure the write port address and data inputs hold values from the same instruction. Otherwise, the write back would be writing the values for the instruction in the write-back stage, but with registers IDs from the instruction in the decode stage. As a general principle, we want to keep all of the information about a particular instruction contained within a single pipeline stage.

One block of PIPE– that is not present in SEQ+ in the exact same form is the block labeled "Select A" in the decode stage. We can see that this block generates the value valA for the pipeline register E by choosing either valP from pipeline register D or the value read from the A port of the register file. This block is included to reduce the amount of state that must be carried forward to pipeline registers E and M. Of all the different instructions, only the `call` requires valP in the memory stage. Only the jump instructions require the value of valP in the execute stage (in the event the jump is not taken). None of these instructions requires a value read from the register file. Therefore we can reduce the amount of pipeline register state by merging these two signals and carrying them through the pipeline as a single signal valA. This eliminates the need for the block labeled "Data" in SEQ (Figure 21) and SEQ+ (Figure 31), which served a similar purpose. In hardware design, it is common to carefully identify how signals get used and then reduce the amount of register state and wiring by merging signals such as these.

5.3 Next PC Prediction

We have taken some measures in the design of PIPE– to properly handle control dependencies. Our goal in the pipelined design is to *issue* a new instruction on every clock cycle, meaning that on each clock cycle, a new instruction proceeds into the execute stage and will ultimately be completed. Achieving this goal would yield a throughput of one instruction per cycle. To do this, we must determine the location of the next instruction right after fetching the current instruction. Unfortunately, if the fetched instruction is a conditional branch, we will not know whether or not the branch should be taken until several cycles later, after the instruction has passed through the execute stage. Similarly, if the fetched instruction is a `ret`, we cannot determine the return location until the instruction has passed through the memory stage.

With the exception of conditional jump instructions and `ret`, we can determine the address of the next instruction based on information computed during the fetch stage. For `call` and `jmp` (unconditional jump), it will be valC, the constant word in the instruction, while for all others it will be valP, the address of the next instruction. We can therefore achieve our goal of issuing a new instruction every clock cycle in most cases by *predicting* the next value of the PC. For most instructions types, our prediction will be completely reliable. For conditional jumps, we can predict either that a jump will be taken, so that the new PC value would be valC, or we can predict that it will not be taken, so that the new PC value would be valP. In either case, we must somehow deal with the case where our prediction was incorrect, and therefore we have fetched and partially executed the wrong instructions. We will return to this matter in Section 5.9.

This technique of guessing the branch direction and then initiating the fetching of instructions according to our guess is known as *branch prediction*. It is used in some form by virtually all processors. Extensive studies have be done on effective strategies for predicting whether or not branches will be taken [3]. Some

systems devote large amounts of hardware to this task. In our design, we will use the simple strategy of predicting that conditional branches are always taken, and so we predict the new value of the PC to be valC.

Aside: Other Branch Prediction Strategies
Our design uses an *always taken* branch prediction strategy. Studies show this strategy has around a 60% success rate [3]. Conversely, a *never taken* (NT) strategy has around a 40% success rate. A slightly more sophisticated strategy, known as *backward taken, forward not-taken* (BTFNT), predicts that branches to lower addresses than the next instruction will be taken while those to higher addresses will not be taken. This strategy has around a 65% success rate. This improvement stems from the fact that loops are closed by backward branches, and loops are generally executed multiple times. Forward branches are used for conditional operations, and these are less likely to be taken. In homework problems 39 and 40 you can modify the Y86 pipeline processor to implement the NT and BTFNT branch prediction strategies.

The effect of unsuccessful branch prediction on program performance is discussed in the context of program optimization in Section **??**. **End Aside.**

We are still left with predicting the new PC value resulting from a `ret` instruction. Unlike conditional jumps, we have a nearly unbounded set of possible results, since the return address will be whatever word is on the top of the stack. In our design, we will not attempt to predict any value for the return address. Instead, we will simply hold off processing any more instructions until the `ret` instruction passes through the write-back stage. We will return to this part of the implementation in Section 5.9.

Aside: Return Address Prediction with a Stack
With most programs, it is very easy to predict return addresses, since procedure calls and returns occur in matched pairs. Most of the time that a procedure is called, it returns to the instruction following the call. This property is exploited in high-performance processors by including a hardware stack within the instruction fetch unit that holds the return address generated by procedure call instructions. Every time a procedure call instruction is executed, its return address is pushed onto the stack. When a return instruction is fetched, the top value is popped from this stack and used as the predicted return address. Like branch prediction, a mechanism must be provided to recover when the prediction was incorrect, since there are times when calls and returns do not match. In general, the prediction is highly reliable. This hardware stack is not part of the programmer-visible state. **End Aside.**

The PIPE– fetch stage, diagrammed at the bottom of Figure 41, is responsible for both predicting the next value of the PC and for selecting the actual PC for the instruction fetch. We can see the block labeled "Predict PC" can choose either valP, as computed by the PC incrementer or valC, from the fetched instruction. This value is stored in pipeline register F as the *predicted* value of the program counter. The block labeled "Select PC" is similar to the block labeled "PC" in the SEQ+ PC selection stage (Figure 31). It chooses one of three values to serve as the address for the instruction memory: the predicted PC, the value of valP for a not-taken branch instruction that reaches pipeline register M (stored in register M_valA), or the value of the return address when a `ret` instruction reaches pipeline register W (stored in W_valM).

We will return to the handling of jump and return instructions when we complete the pipeline control logic in Section 5.9.

5.4 Pipeline Hazards

Our structure PIPE– is a good start at creating a pipelined Y86 processor. Recall from our discussion in Section 4.4, however, that introducing pipelining into a system with feedback can lead to problems when

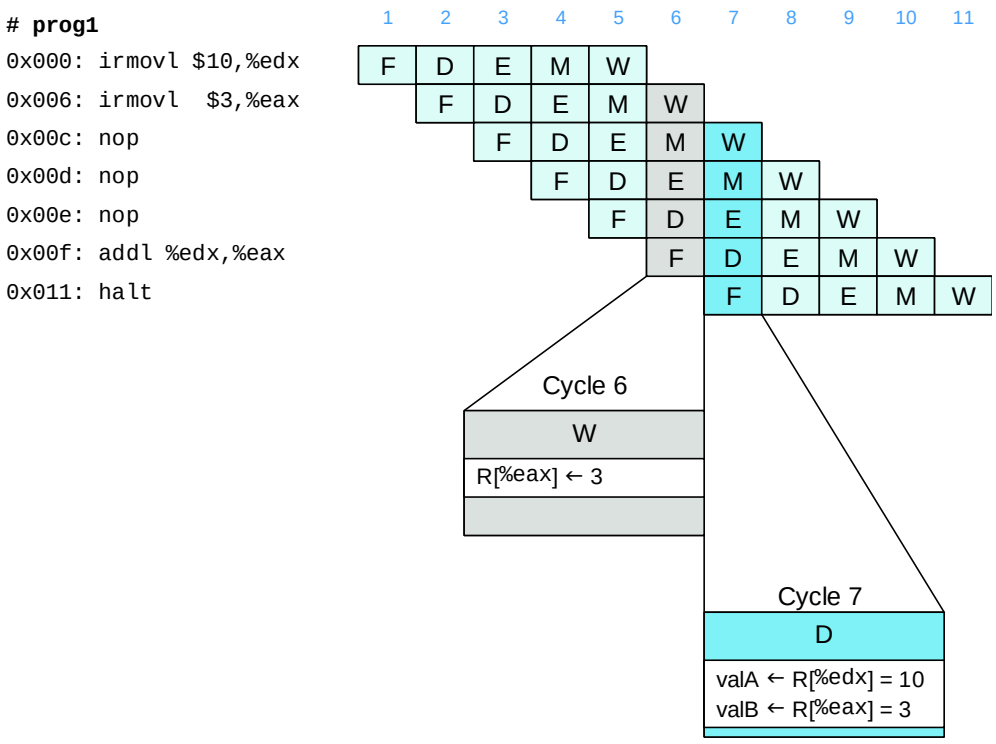


Figure 42: **Pipelined execution of `prog1` without special pipeline control.** In cycle 6, the second `irmovl` writes its result to program register `%eax`. The `addl` instruction reads its source operands in cycle 7, so it gets correct values for both `%edx` and `%eax`.

there are dependencies between successive instructions. We must resolve this issue before we can complete our design. These dependencies can take two forms: (1) *data* dependencies, where the results computed by one instruction are used as the data for a following instruction, and (2) *control* dependencies, where one instruction determines the location of the following instruction, such as when executing a jump, call, or return. When such dependencies have the potential to cause an erroneous computation by the pipeline, they are called *hazards*. Like dependencies, hazards can be classified as either *data hazards* or *control hazards*. In this section, we concern ourselves with data hazards. Control hazards will be discussed as part of overall pipeline control (Section 5.9).

Figure 42 illustrates the processing of a sequence of instructions we refer to as `prog1` by the PIPE– processor. This code loads values 10 and 3 into program registers `%edx` and `%eax`, executes three `nop` instructions, and then adds register `%edx` to `%eax`. We focus our attention on the potential data hazards resulting from the data dependencies between the two `irmovl` instructions and the `addl` instruction. On the right-hand side of the figure, we show a pipeline diagram for the instruction sequence. The pipeline stages for cycles 6 and 7 are shown highlighted in the pipeline diagram. Below this we show an expanded view of the write-back activity in cycle 6 and the decode activity during cycle 7. Before the start of cycle 7, both of the `irmovl` instructions have passed through the write-back stage, and so the register file holds the updated values of `%edx` and `%eax`. As the `addl` instruction passes through the decode stage during cycle 7, it will therefore read the correct values for its source operands. The data dependencies between the two `irmovl`

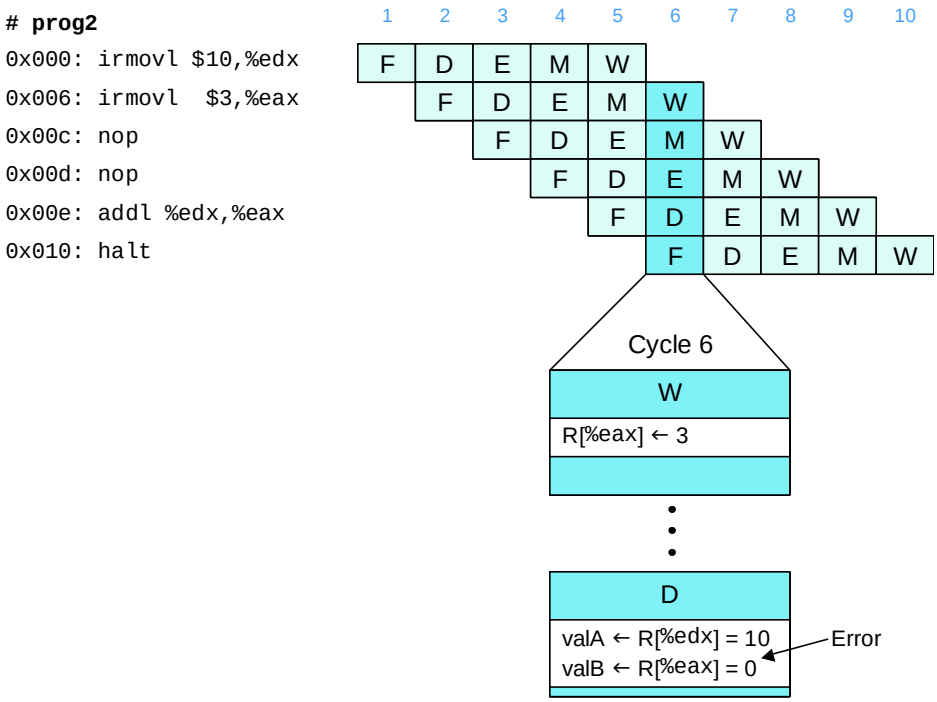


Figure 43: **Pipelined execution of `prog2` without special pipeline control.** The write to program register `%eax` does not occur after the end of cycle 6, causing the `addl` instruction to get the incorrect value for this register in the decode stage.

instructions and the `addl` instruction have not created data hazards in this example.

We saw that `prog1` will flow through our pipeline and get the correct results, because the three `nop` instructions create a delay between instructions with data dependencies. Let's see what happens as these `nop` instructions are removed. Figure 43 illustrates the pipeline flow of a program, named `prog2`, containing two `nop` instruction between the two `irmovl` instructions generating values for registers `%edx` and `%eax`, and the `addl` instruction having these two registers as operands. In this case, the crucial step occurs in cycle 6, when the `addl` instruction reads its operands from the register file. An expanded view of the pipeline activities during this cycle is shown at the bottom of the figure. The first `irmovl` instruction has passed through the write-back stage, and so program register `%edx` has been updated in the register file. The second `irmovl` instruction is in the write-back stage during this cycle, and so the write to program register `%eax` only occurs at the start of cycle 7 as the clock rises. As a result, the incorrect value would be read for register `%eax` (here we assume all registers are initially 0), since the pending write for this register has not yet occurred. Clearly we will have to adapt our pipeline to handle this hazard properly.

Figure 44 shows what happens when we have only one `nop` instructions between the `irmovl` instructions and the `addl` instruction, yielding a program `prog3`. Now we must examine the behavior of the pipeline during cycle 5 as the `addl` instruction passes through the decode stage. Unfortunately, the pending write to register `%edx` is still in the write-back stage, and the pending write to `%eax` is still in the memory stage. Therefore, the `addl` instruction would get the incorrect values for both operands.

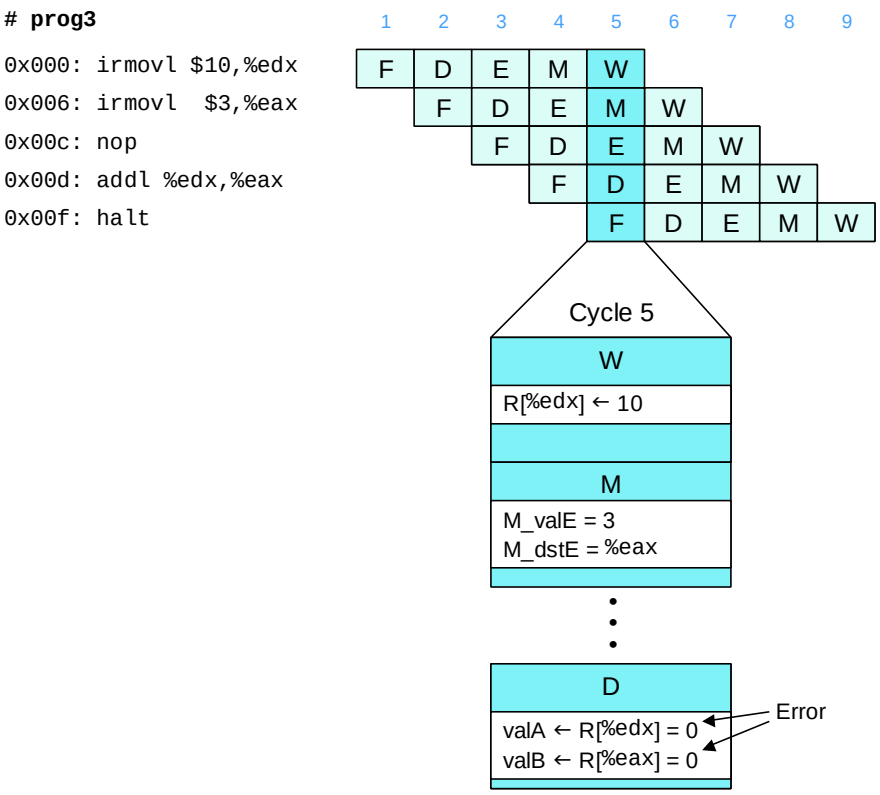


Figure 44: **Pipelined execution of `prog3` without special pipeline control.** In cycle 5, the `addl` instruction reads its source operands from the register file. The pending write to register `%edx` is still in the write-back stage, and the pending write to register `%eax` is still in the memory stage. Both operands valA and valB get incorrect values.

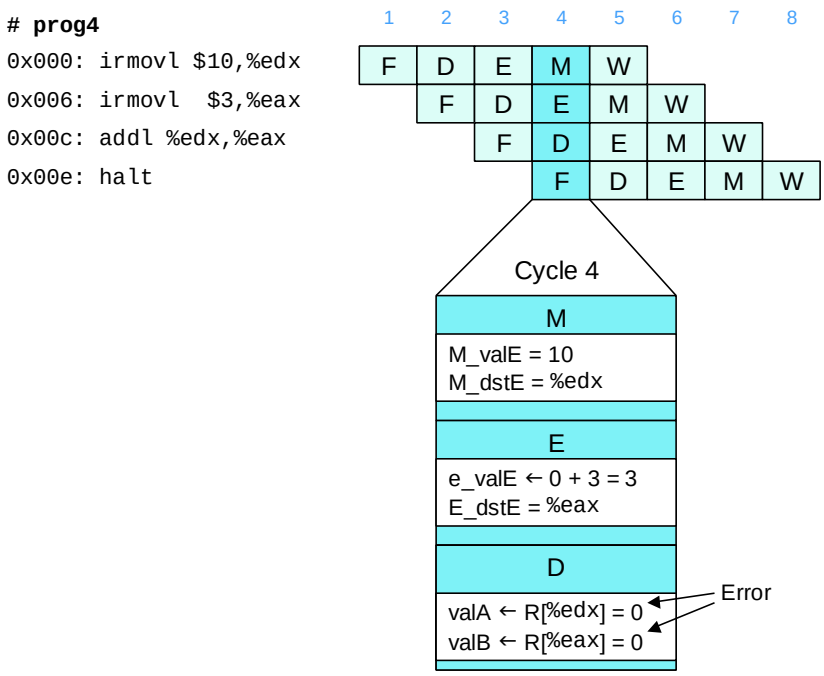


Figure 45: **Pipelined execution of `prog4` without special pipeline control.** In cycle 4, the `addl` instruction reads its source operands from the register file. The pending write to register `%edx` is still in the memory stage, and the new value for register `%eax` is just being computed in the execute stage. Both operands valA and valB get incorrect values.

Figure 45 shows what happens when we remove all of the `nop` instructions between the `irmovl` instructions and the `addl` instruction, yielding a program `prog4`. Now we must examine the behavior of the pipeline during cycle 4 as the `addl` instruction passes through the decode stage. Unfortunately, the pending write to register `%edx` is still in the memory stage, and the new value for `%eax` is just being computed in the execute stage. Therefore the `addl` instruction would get the incorrect values for both operands.

These examples illustrate that a data hazard can arise for an instruction when one of its operands is updated by any of the three preceding instructions. These hazards occur because our pipelined processor reads the operands for an instruction from the register file in the decode stage, but it does not write the results for the instruction to the register file until three cycles later, after the instruction passes through the write-back stage.

Aside: Enumerating Classes of Data Hazards
Hazards can potentially occur when one instruction updates part of the program state that will be read by a later instruction. The program state includes the program registers, the condition codes, the memory, and the program counter. Let's look at the hazard possibilities for each of these forms of state.

Program registers: These are the hazards we have already identified. They arise because the register file is read in one stage and written in another, leading to possible unintended interactions between different instructions.

Condition codes: These are both written (by integer operations) and read (by conditional jumps) in the execute stage. By the time a conditional jump passes through this stage, any preceding integer operations instructions have already completed this stage. No hazards can arise.

Program counter: Conflicts between updating and reading the program counter give rise to control hazards. No hazard arises when our fetch-stage logic correctly predicts the new value of the program counter before fetching the next instruction. Mispredicted branches and `ret` instructions require special handling, as will be discussed in Section 5.9.

Memory: Writes and reads of the data memory both occur in the memory stage. By the time an instruction reading memory reaches this stage, any preceding instructions writing memory will have already done so. On the other hand, there can be interference between instructions writing data in the memory stage, and the reading of instructions in the fetch stage, since the instruction and data memories reference a single address space. This can only happen with programs containing *self-modifying code*, where instructions write to a portion of memory from which instructions are later fetched. Some systems have complex mechanisms to detect and avoid such hazards, while others simply mandate that programs should not use self-modifying code. We will assume for simplicity that programs do not modify themselves.

This analysis shows that we only need to deal with register data hazards and control hazards. **End Aside.**

5.5 Avoiding Data Hazards by Stalling

One very general technique for avoiding hazards involves *stalling*, where the processor holds back one or more instructions in the pipeline until the hazard condition no longer holds. A processor can avoid data hazards by holding back an instruction in the decode stage as long as one of its source operands will be generated by some instruction in a later stage of the pipeline. This technique is diagrammed in Figures 46 (`prog2`), 47 (`prog3`), and 48 (`prog4`). When the `addl` instruction is in the decode stage, the pipeline control logic detects that at least one of the instructions in the execute, memory, or write-back stage will update either register `%edx` or register `%eax`. Rather than letting the `addl` instruction pass through the stage with the incorrect results, it stalls the instruction, holding it back in the decode stage for either one (for `prog2`), two (for `prog3`), or even three (for `prog3`) extra cycles. For all three programs, the `addl`

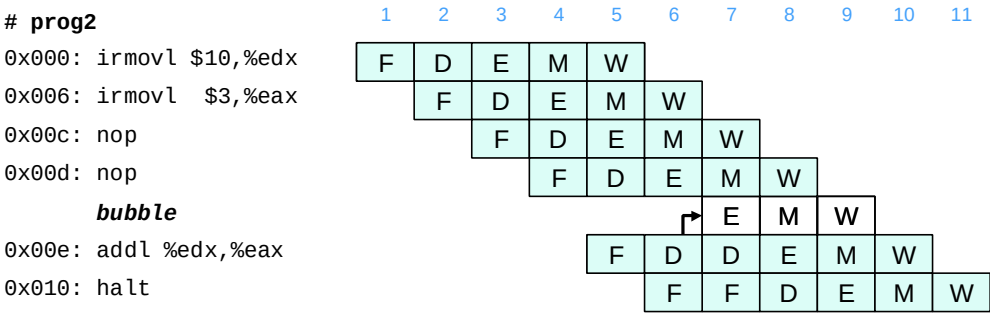


Figure 46: **Pipelined execution of `prog2` using stalls.** After decoding the `addl` instruction in cycle 6, the stall control logic detects a data hazard due to the pending write to register `%eax` in the write-back stage. It injects a bubble into execute stage and repeats the decoding of the `addl` instruction in cycle 7. In effect, the machine has dynamically inserted a `nop` instruction, giving a flow similar to that shown for `prog1` (Figure 42).

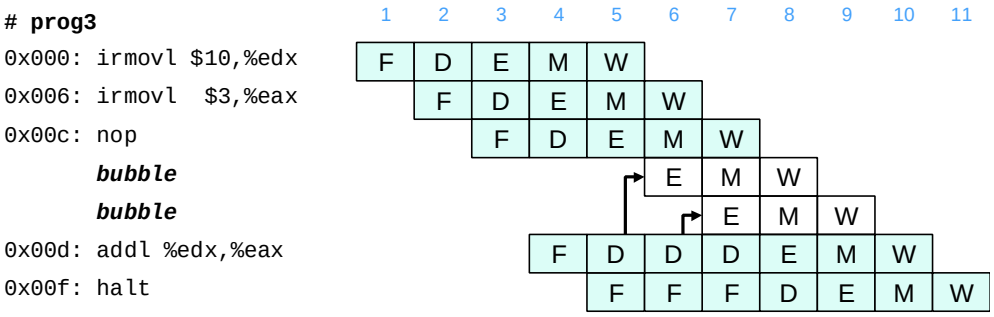


Figure 47: **Pipelined execution of `prog3` using stalls.** After decoding the `addl` instruction in cycle 5, the stall control logic detects data hazards for both source registers. It injects a bubble into the execute stage and repeats the decoding of the `addl` instruction on cycle 6. It again detects a hazard for register `%eax`, injects a second bubble into the execute stage, and repeats the decoding of the `addl` instruction in cycle 7. In effect, the machine has dynamically inserted two `nop` instructions, giving a flow similar to that shown for `prog1` (Figure 42).

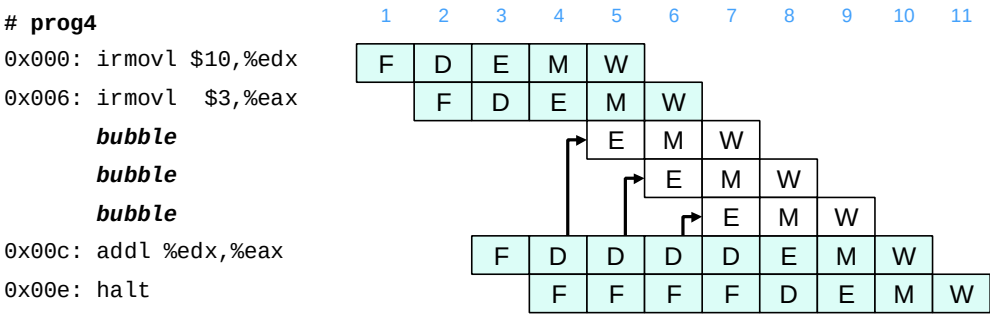


Figure 48: **Pipelined execution of `prog4` using stalls.** After decoding the `addl` instruction in cycle 4, the stall control logic detects data hazards for both source registers. It injects a bubble into the execute stage and repeats the decoding of the `addl` instruction on cycle 5. It again detects hazards for both source registers, injects a bubble into the execute stage, and repeats the decoding of the `addl` instruction on cycle 6. Still, it detects a hazard for source register `%eax`, injects a bubble into the execute stage, and repeats the decoding of the `addl` instruction on cycle 7. In effect, the machine has dynamically inserted three `nop` instructions, giving a flow similar to that shown for `prog1` (Figure 42).

instruction finally gets correct values for its two source operands in cycle 7 and then proceeds down the pipeline.

In holding back the `addl` instruction in the decode stage, we must also hold back the `halt` instruction following it in the fetch stage. We can do this by keeping the program counter at a fixed value, so that the `halt` instruction will be fetched repeatedly until the stall has completed.

Stalling involves holding back one group of instructions in their stages while allowing other instructions to continue flowing through the pipeline. In our example, we hold back the `addl` in the decode stage and the `halt` in the fetch stage for one to three extra cycles, while letting the two `irmovl` instructions and the `nop` instructions (in the cases of `prog2` and `prog3`) continue through the execute, memory, and write-back stages. What then should we do in the stages that would normally be processing the `addl` instruction? We handle these by injecting a *bubble* into the execute stage each time we hold an instruction back in the decode stage. A bubble is like a dynamically generated `nop` instruction—it does not cause any changes to the registers, the memory, or the condition codes. These are shown as white boxes in the pipeline diagrams of Figures 46 through 48. In these figures we show arrows between one of the boxes labeled "D" for the `addl` instruction, to a box labeled "E" for one of the pipeline bubbles. These arrows indicate that a bubble was injected into the execute stage in place of the `addl` instruction that would normally have passed from the decode to the execute stage. We will look at the detailed mechanisms for making the pipeline stall and for injecting bubbles in Section 5.9.

In using stalling to handle data hazards, we effectively execute programs `prog2`, `prog3`, and `prog4` by dynamically generating the pipeline flow seen for `prog1` (Figure 42). Injecting one bubble for `prog2`, two for `prog3`, and three for `prog4` has the same effect as having three `nop` instructions between the second `irmovl` instruction and the `addl` instruction. This mechanism can be implemented fairly easily (see homework problem 36), but the resulting performance is not very good. There are numerous cases in which one instruction updates a register and a closely following instruction uses the same register. This will cause the pipeline to stall for up to three cycles, reducing the overall throughput significantly.

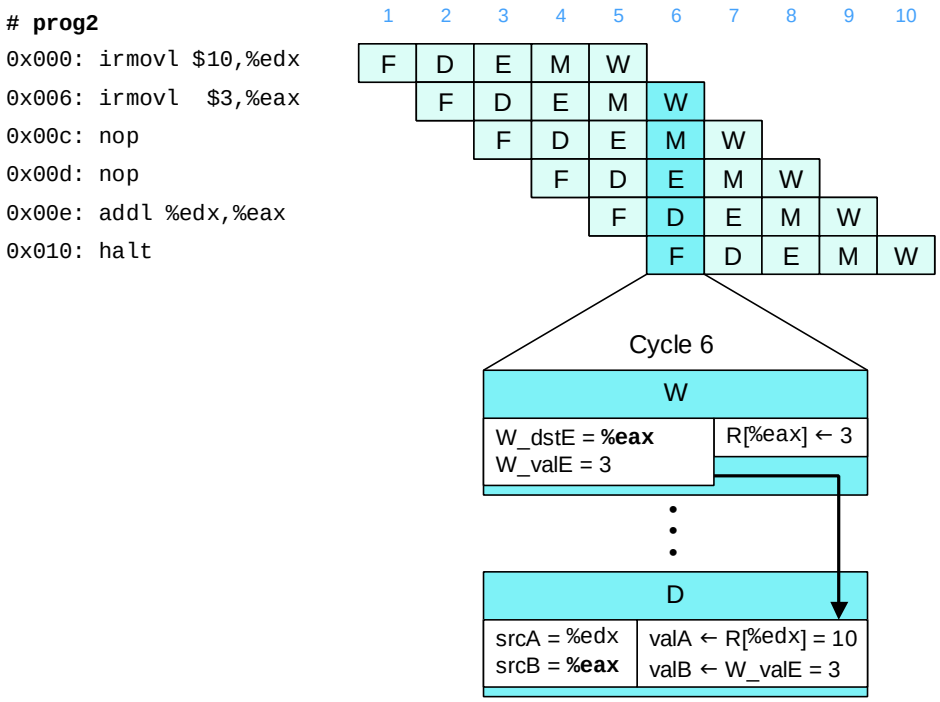


Figure 49: **Pipelined execution of `prog2` using forwarding.** In cycle 6, the decode-stage logic detects the presence of a pending write to register `%eax` in the write-back stage. It uses this value for source operand valB, rather than the value read from the register file.

5.6 Avoiding Data Hazards by Forwarding

Our design for PIPE– reads source operands from the register file in the decode stage, but there can also be a pending write to one of these source registers in the write-back stage. Rather than stalling until the write has completed, it can simply pass the value that is about to be written to pipeline register E as the source operand. Figure 49 shows this strategy with an expanded view of the pipeline diagram for cycle 6 of `prog2`. The decode-stage logic detects that register `%eax` is the source register for operand valB, and that there is also a pending write to `%eax` on write port E. It can therefore avoid stalling by simply using the data word supplied to port E (signal W_valE) as the value for operand valB. This technique of passing a result value directly from one pipeline stage to an earlier one is known as *data forwarding* (or simply *forwarding*). It allows the instructions of `prog2` to proceed through the pipeline without any stalling.

As Figure 50 illustrates, data forwarding can also be used when there is a pending write to a register in the memory stage, avoiding the need to stall for program `prog3`. In cycle 5, the decode-stage logic detects a pending write to register `%edx` on port E in the write-back stage, as well as a pending write to register `%eax` that is on its way to port E but is still in the memory stage. Rather than stalling until the writes have occurred, it can use the value in the write-back stage (signal W_valE) for operand valA and the value in the memory stage (signal M_valE) for operand valB.

To exploit data forwarding to its full extent, we can also pass newly computed values from the execute stage to the decode stage, avoiding the need to stall for program `prog4`, as illustrated in Figure 51. In cycle 4,

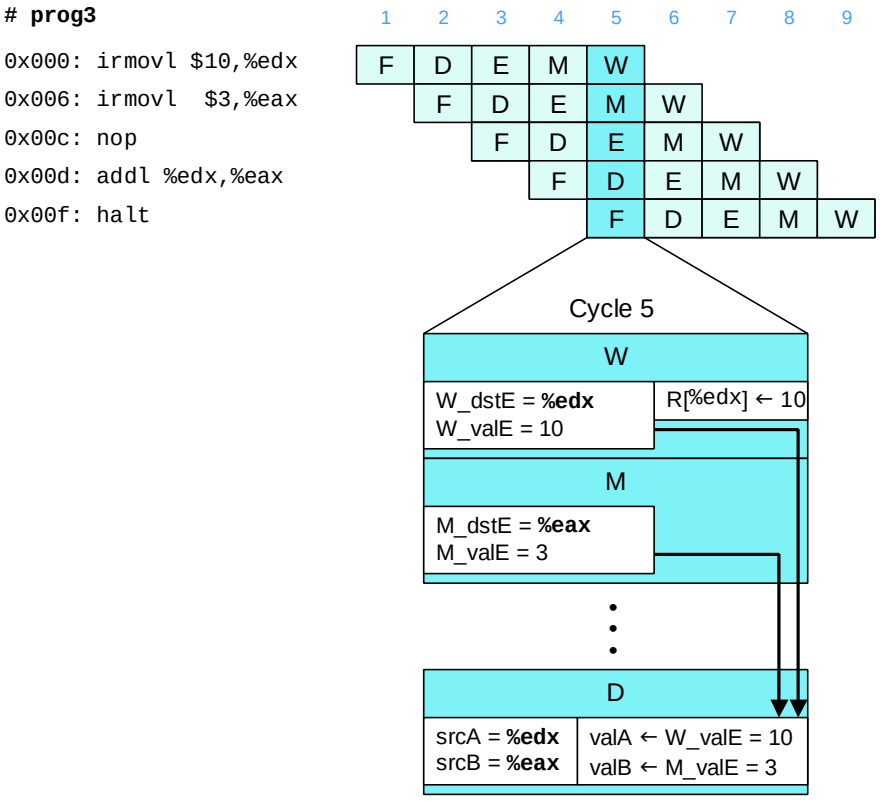


Figure 50: **Pipelined execution of `prog3` using forwarding.** In cycle 5, the decode-stage logic detects a pending write to register `%edx` in the write-back stage and to register `%eax` in the memory stage. It uses these as the values for valA and valB rather than the values read from the register file.

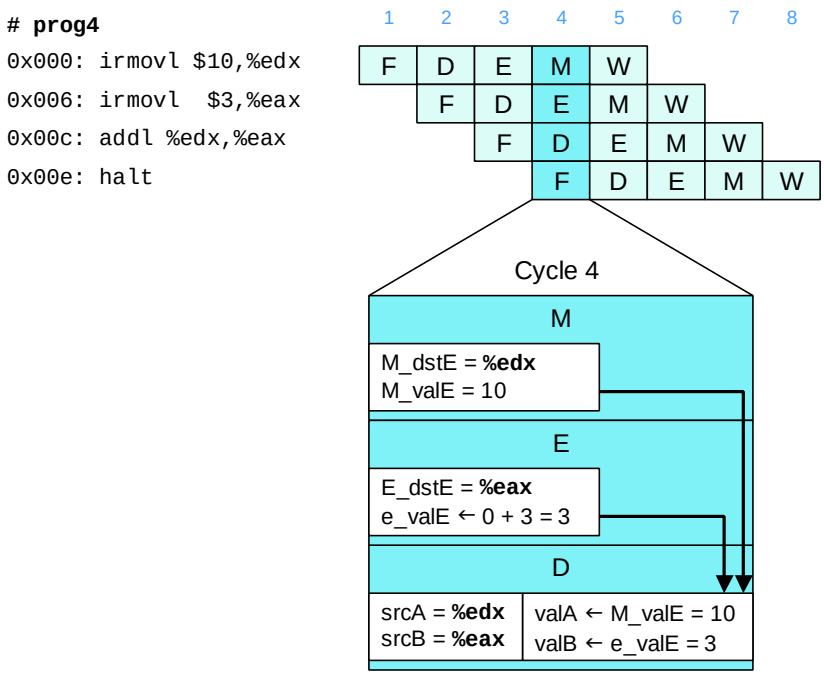


Figure 51: **Pipelined execution of `prog4` using forwarding.** In cycle 4, the decode-stage logic detects a pending write to register `%edx` in the memory stage. It also detects that a new value is being computed for register `%eax` in the execute stage. It uses these as the values for valA and valB rather than the values read from the register file.

the decode-stage logic detects a pending write to register %edx in the memory stage, and also that the value being computed by the ALU in the execute stage will later be written to register %eax. It can use the value in the memory stage (signal M_valE) for operand valA. It can also use the ALU output (signal e_valE) for operand valB. Note that using the ALU output does not introduce any timing problems. The decode stage only needs to generate signals valA and valB by the end of the clock cycle so that pipeline register E can be loaded with the results from the decode stage as the clock rises to start the next cycle. The ALU output will be valid before this point.

The uses of forwarding illustrated in programs prog2 to prog4 all involve the forwarding of values generated by the ALU and destined for write port E. Forwarding can also be used with values read from the memory and destined for write port M. From the memory stage, we can forward the value that has just been read from the data memory (signal m_valM). From the write-back stage, we can forward the pending write to port M (signal W_valM). This gives a total of five different forwarding sources (e_valE, m_valM, M_valE, W_valM, and W_valE), and two different forwarding destinations (valA and valB).

The expanded diagrams of Figures 49 to 51 also show how the decode-stage logic can determine whether to use a value from the register file or to use a forwarded value. Associated with every value that will be written back to the register file is the destination register ID. The logic can compare these IDs with the source register IDs srcA and srcB, to detect a case for forwarding. It is possible to have multiple destination register IDs match one of the source IDs. We must establish a priority among the different forwarding sources to handle such cases. This will be discussed when we look at the detailed design of the forwarding logic.

Figure 52 shows the abstract structure of PIPE, an extension of PIPE– that can handle data hazards by forwarding. We can see that additional feedback paths (shown in blue) have been added from the five forwarding sources down to the decode stage. These *bypass paths* feed into a block labeled "Forward" in the decode stage. This block generates the source operands valA and valB using either values read from the register file or one of the forwarded values.

Figure 53 gives a more detailed view of the PIPE hardware structure. Comparing this to the structure of PIPE– (Figure 41), we can see that the values from the five forwarding sources are fed back to the two blocks labeled "Sel+Fwd A" and "Fwd B" in the decode stage. The block labeled "Sel+Fwd A" combines the role of the block labeled "Select A" in PIPE– with the forwarding logic. It allows valA for pipeline register M to be either the incremented program counter valP, the value read from the A port of the register file, or one of the forwarded values. The block labeled "Fwd B" implements the forwarding logic for source operand valB.

5.7 Load/Use Data Hazards

One class of data hazards cannot be handled purely by forwarding, because memory reads occur late in the pipeline. Figure 54 illustrates an example of a *load/use hazard*, where one instruction (the mrmovl at address 0x018) reads a value from memory for register %eax, while the next instruction (the addl at address 0x01e) needs this value as a source operand. Expanded views of cycles 7 and 8 are shown in the lower part of the figure. The addl instruction requires the value of the register in cycle 7, but it isn't generated by the mrmovl instruction until cycle 8. In order to "forward" from the mrmovl to the addl, the forwarding logic would have to make the value go backward in time! Since this is clearly impossible,

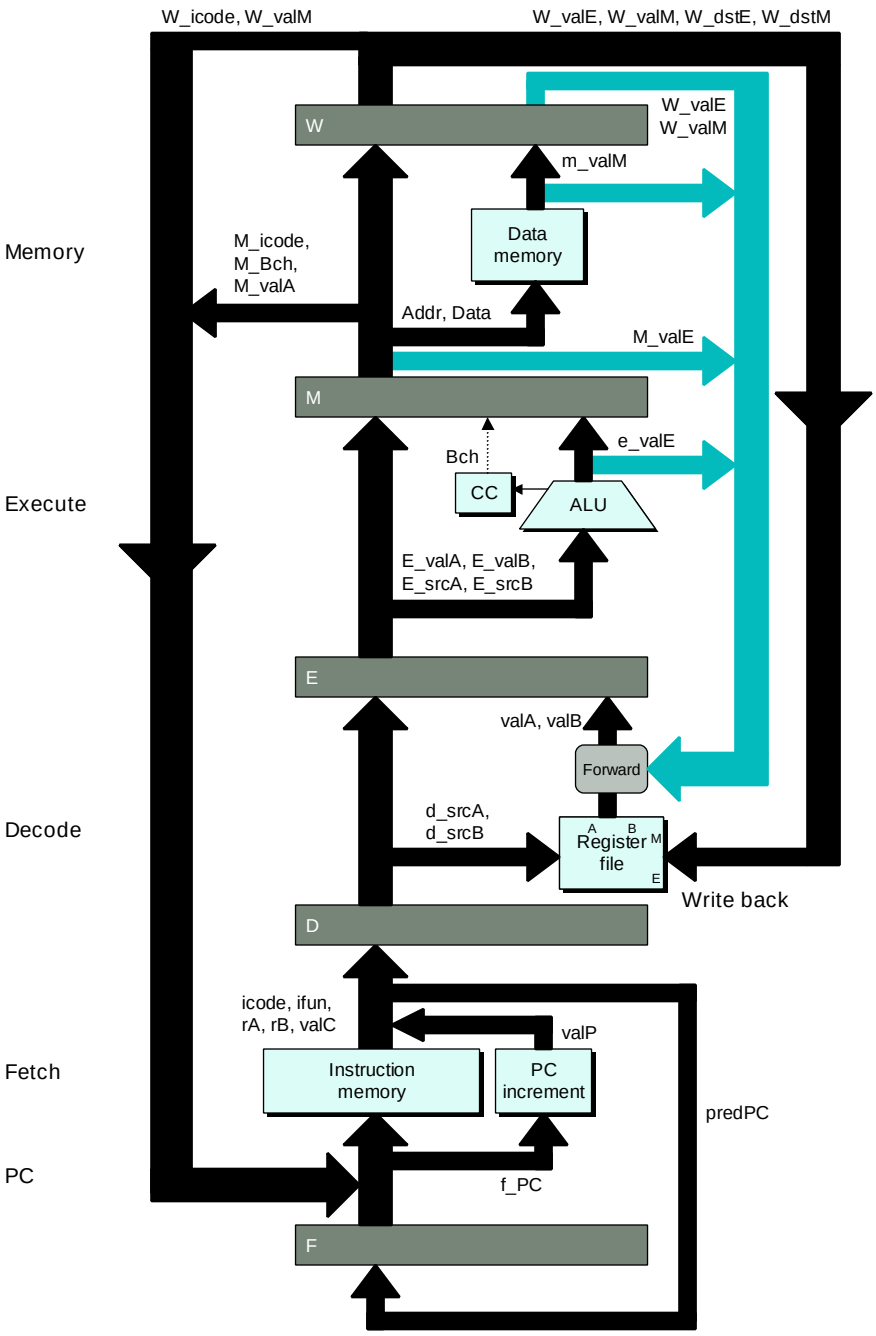


Figure 52: **Abstract view of PIPE, our final pipelined implementation.** The additional bypassing paths (shown in blue) enable forwarding the results from the three preceding instructions. This allows us to handle most forms of data hazards without stalling the pipeline.

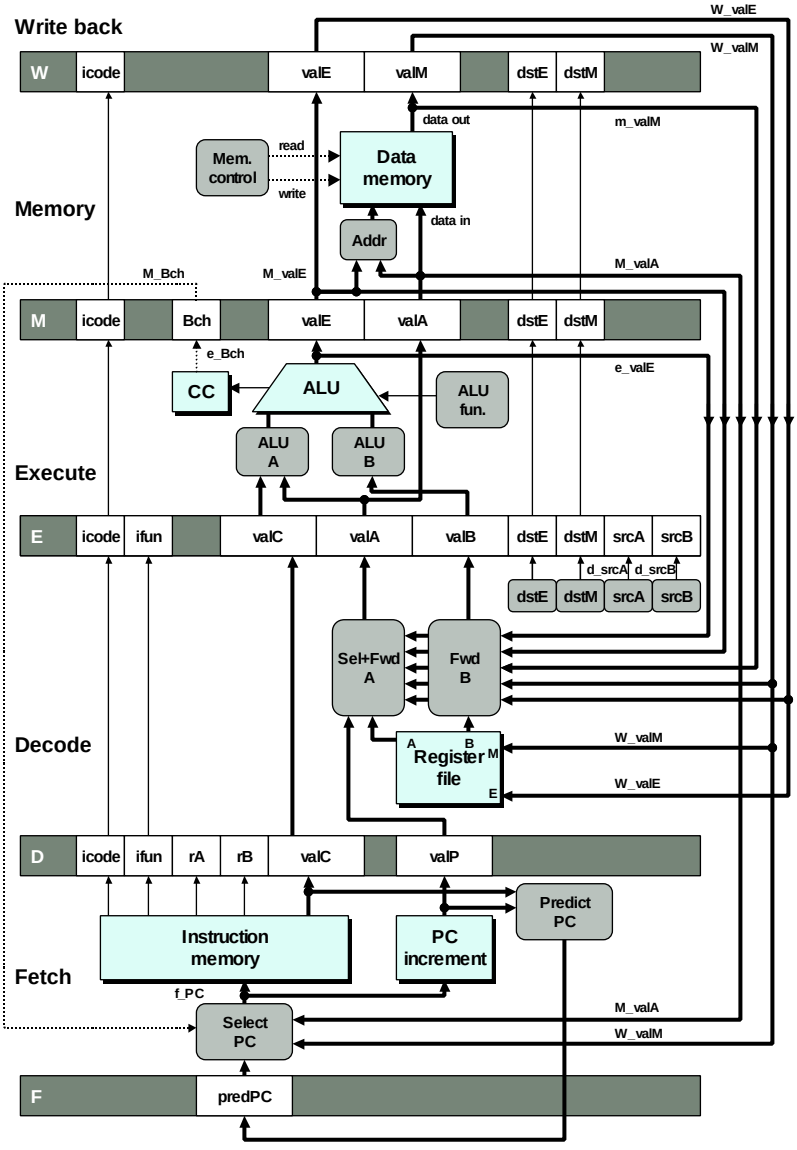


Figure 53: **Hardware structure of PIPE, our final pipelined implementation.** Some of the connections are not shown.

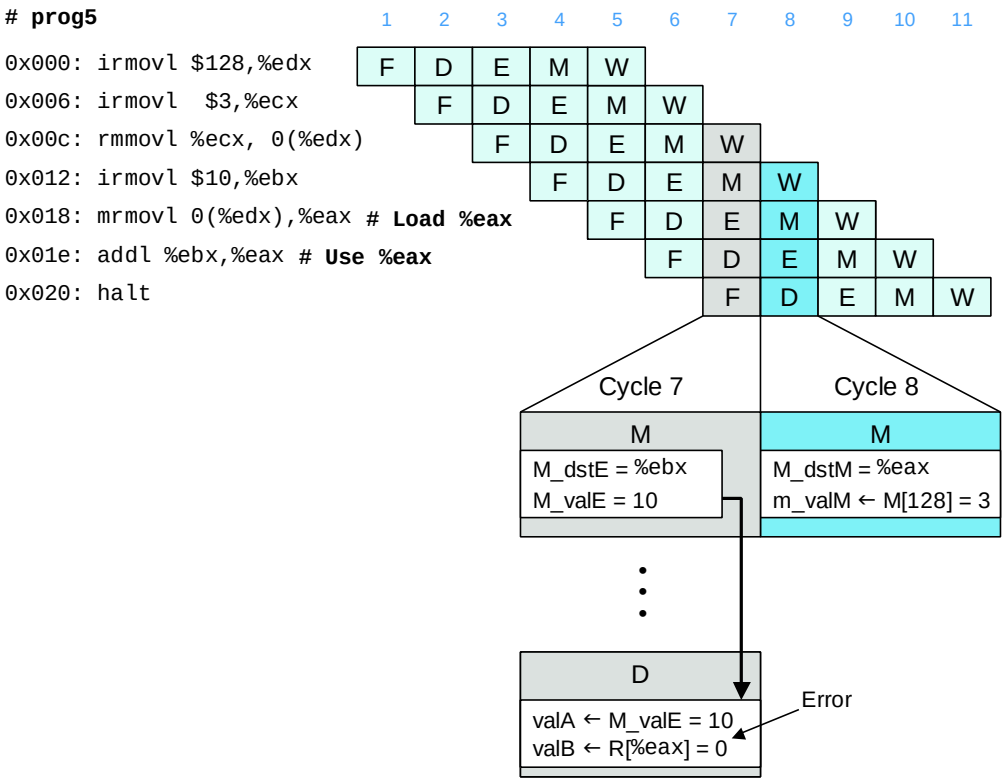


Figure 54: **Example of load/use data hazard.** The `addl` instruction requires the value of register `%eax` during the decode stage in cycle 7. The preceding `mrmovl` reads a new value for this register `%eax` during the memory stage in cycle 8, which is too late for the `addl` instruction.

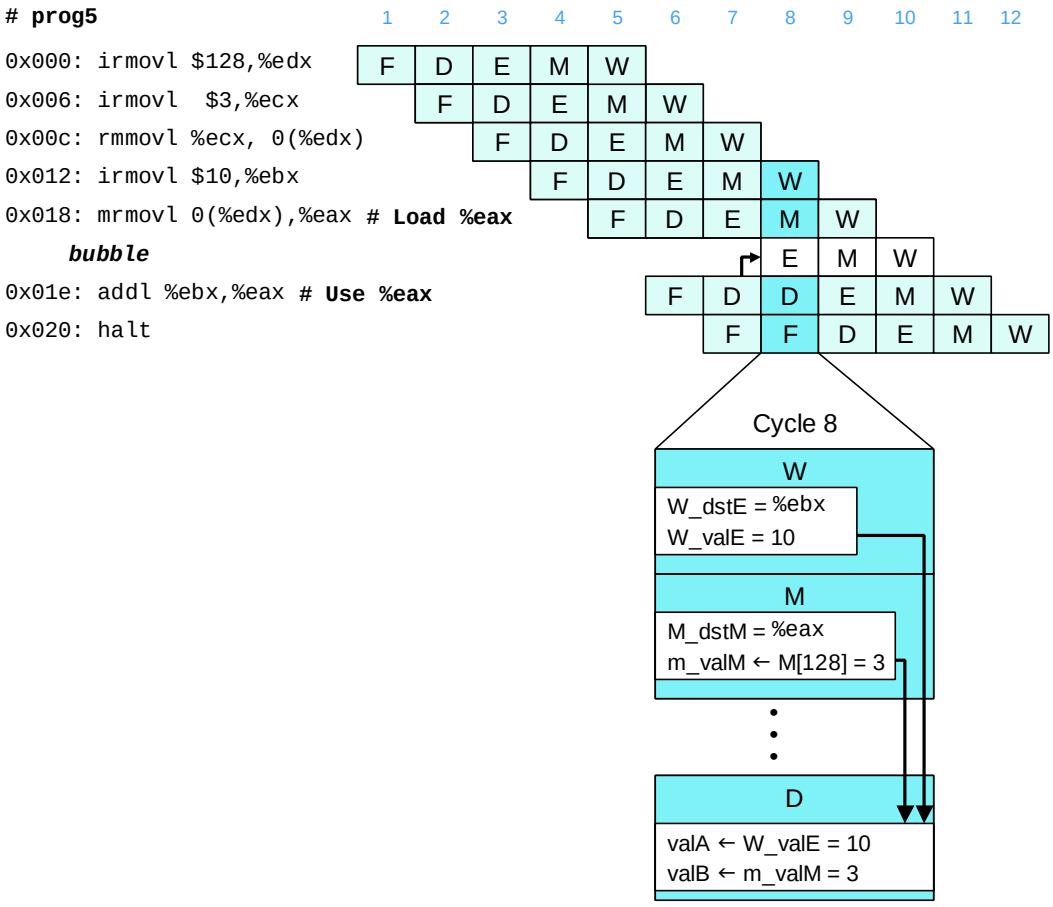


Figure 55: **Handling a load/use hazard by stalling.** By stalling the `addl` instruction for one cycle in the decode stage, the value for valB can be forwarded from the `mrmovl` instruction in the memory stage to the `addl` instruction in the decode stage.

we must find some other mechanism for handling this form of data hazard. Note that the value for register `%ebx`, generated by the `irmovl` instruction at address `0x00c`, can be forwarded from the memory stage to the `addl` instruction in its decode stage on cycle 7.

As Figure 55 demonstrates, we can avoid a load/use data hazard with a combination of stalling and forwarding. As the `mrmovl` instruction passes through the execute stage, the pipeline control logic detects that the instruction in the decode stage (the `addl`) requires the result read from memory. It stalls the instruction in the decode stage for one cycle, causing a bubble to be injected into the execute stage. As the expanded view of cycle 8 shows, the value read from memory can then be forwarded from the memory stage to the `addl` instruction in the decode stage. The value for register `%edx` is also forwarded from the write-back to the memory stage. As indicated in the pipeline diagram by the arrow from the box labeled "D" in cycle 7 to the box labeled "E" in cycle 8, the injected bubble replaces the `addl` instruction that would normally continue flowing through the pipeline.

This use of a stall to handle a load/use hazard is called a *load interlock*. Load interlocks combined with

forwarding suffice to handle all possible forms of data hazards. Since only load interlocks reduce the pipeline throughput, we can nearly achieve our throughput goal of issuing one new instruction on every clock cycle.

5.8 PIPE Stage Implementations

We have now created an overall structure for PIPE, our pipelined Y86 processor with forwarding. It uses the same set of hardware units as the earlier sequential designs, with the addition of pipeline registers, some reconfigured logic blocks, and additional pipeline control logic. In this section we go through the design the different logic blocks, deferring the design of the pipeline control logic to the next section. Many of the logic blocks are identical to their counterparts in SEQ and SEQ+, except that we must choose proper versions of the different signals from the pipeline registers (written with the pipeline register name, written in upper case, as a prefix) or from the stage computations (written with the first character of the stage name, written in lower case, as a prefix).

As an example, compare the HCL code for the logic that generates the srcA signal in SEQ to the corresponding code in PIPE:

```
# Code from SEQ

int srcA = [
        icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : rA;
        icode in { IPOPL, IRET } : RESP;
        1 : RNONE; # Don't need register
];

# Code from PIPE

int new_E_srcA = [
        D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : D_rA;
        D_icode in { IPOPL, IRET } : RESP;
        1 : RNONE; # Don't need register
];
```

They differ only by the prefix "D_" added to the PIPE signals to indicate that the signals come from pipeline register D. To avoid repetition, we will not show the HCL code here for blocks that only differ from those in SEQ because of the prefixes on names. As a reference, though, the complete HCL code for PIPE is given in Section D of Appendix 6.1.

PC Selection and Fetch Stage

Figure 56 provides a detailed view of the PIPE fetch stage logic. As discussed earlier, this stage must also select a current value for the program counter and predict the next PC value. The hardware units for reading the instruction from memory and for extracting the different instruction fields are the same as those we considered for SEQ (see the fetch stage in Section 3.4).

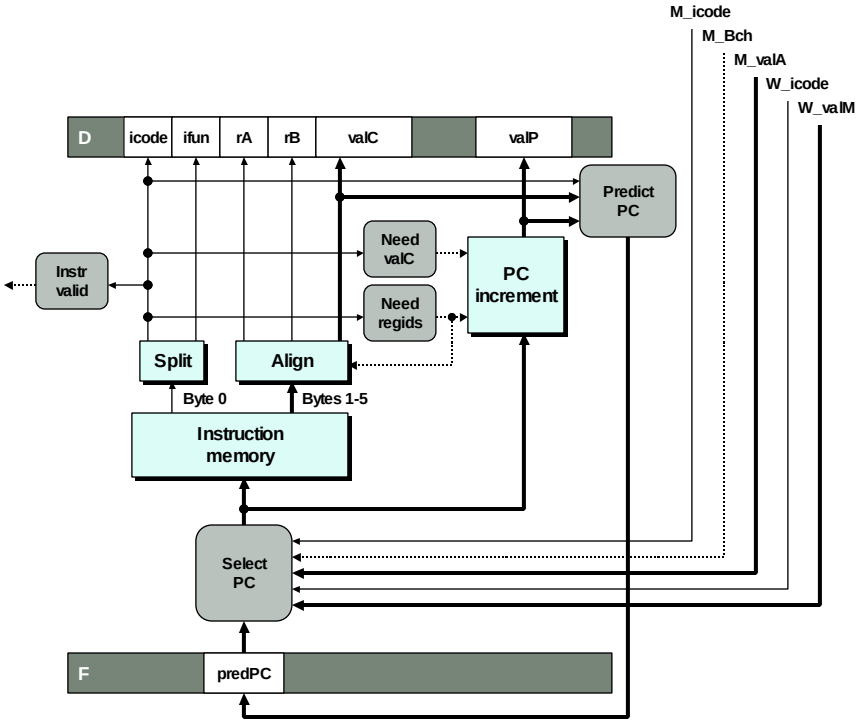


Figure 56: **PIPE PC selection and fetch logic.** Within the one cycle time limit, the processor can only predict the address of the next instruction.

The PC selection logic chooses between three program counter sources. As a mispredicted branch enters the memory stage, the value of valP for this instruction (indicating the address of the following instruction) is read from pipeline register M (signal M_valA). When a `ret` instruction enters the write-back stage, the return address is read from pipeline register W (signal W_valM). All other cases use the predicted value of the PC, stored in pipeline register F (signal F_predPC):

```
int f_pc = [
        # Mispredicted branch.  Fetch at incremented PC
        M_icode == IJXX && !M_Bch : M_valA;
        # Completion of RET instruction.
        W_icode == IRET : W_valM;
        # Default: Use predicted value of PC
        1 : F_predPC;
];
```

The PC prediction logic chooses valC for the fetched instruction when it is either a call or a jump, and valP otherwise:

```
int new_F_predPC = [
        f_icode in { IJXX, ICALL } : f_valC;
        1 : f_valP;
];
```

The logic blocks labeled “Instr valid,” “Need regids,” and “Need valC” are the same as for SEQ, with appropriately named source signals.

Decode and Write-Back Stage

Figure 57 gives a detailed view of the PIPE decode and write-back logic. The blocks labeled “dstE,” “dstM,” “srcA,” and “srcB” are very similar to their counterparts in the implementation of SEQ. Observe that the register IDs supplied to the write ports come from the write-back stage (signals W_dstE and W_dstM), rather than from the decode stage. This is because we want the writes to occur to the destination registers specified by the instruction in the write-back stage.

> **Practice Problem 23**:
>
> The block labeled “dstE” in the decode stage generates the dstE signal based on fields from the fetched instruction in pipeline register D. The resulting signal is named new_E_dstE in the HCL description of PIPE. Write HCL code for this signal, based on the HCL description of the SEQ signal dstE. (See the decode stage in Section 3.4.)

Most of the complexity of this stage is associated with the forwarding logic. As mentioned earlier, the block labeled “Sel+Fwd A” serves two roles. It merges the valP signal into the valA signal for later stages in order to reduce the amount of state in the pipeline register. It also implements the forwarding logic for source operand valA.

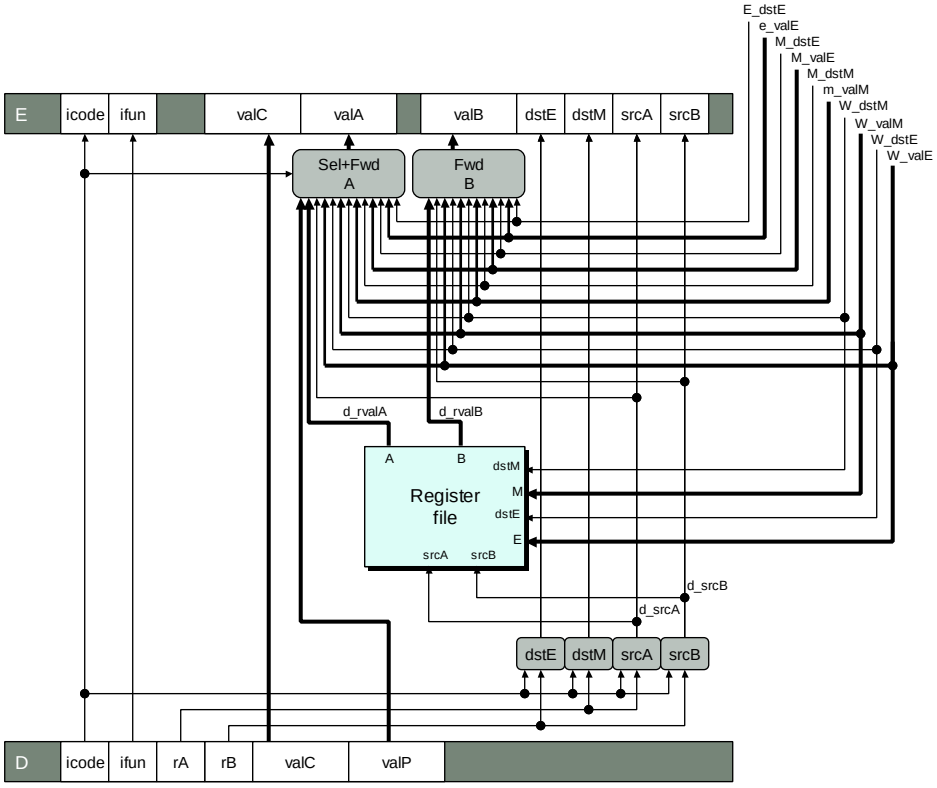


Figure 57: **PIPE decode and write-back stage logic.** No instruction requires both valP and the value read from register port A, and so these two can be merged to form the signal valA for later stages. The block labeled “Sel+Fwd A” performs this task and also implements the forwarding logic for source operand valA. The block labeled “Fwd B” implements the forwarding logic for source operand valB. The register write locations are specified by the dstA and dstB signals from the write-back stage rather than from the decode stage, since it is writing the results of the instruction currently in the write-back stage.

The merging of signals valA and valP exploits the fact that only the `call` and jump instructions need the value of valP in later stages, and these instructions do not need the value read from the A port of the register file. This selection is controlled by the icode signal for this stage. When signal D_icode matches the instruction code for either `call` or `jXX`, this block should select D_valP as its output.

As mentioned in Section 5.6, there are five different forwarding sources, each with a data word and a destination register ID:

Data word	Register ID	Source description
e_valE	E_dstE	ALU output
m_valM	M_dstM	Memory output
M_valE	M_dstE	Pending write to port E in memory stage
W_valM	W_dstM	Pending write to port M in write-back stage
W_valE	W_dstE	Pending write to port E in write-back stage

If none of the forwarding conditions hold, the block should select d_rvalA, the value read from register port A as its output.

Putting all of this together, we get the following HCL description for the new value of valA for pipeline register M:

```
int new_E_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        d_srcA == E_dstE : e_valE;    # Forward valE from execute
        d_srcA == M_dstM : m_valM;    # Forward valM from memory
        d_srcA == M_dstE : M_valE;    # Forward valE from memory
        d_srcA == W_dstM : W_valM;    # Forward valM from write back
        d_srcA == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalA;  # Use value read from register file
];
```

The priority given to the five forwarding sources in the above HCL code is very important. This priority is determined in the HCL code by the order in which the five destination register IDs are tested. If any order other than the one shown were chosen, the pipeline would behave incorrectly for some programs. Figure 58 shows an example of a program that requires a correct setting of priority among the forwarding sources in the execute and memory stages. In this program, the first two instructions write to register `%edx`, while the third uses this register as its source operand. When the `rrmovl` instruction reaches the decode stage in cycle 4, the forwarding logic must choose between two values destined for its source register. Which one should it choose? To set the priority, we must consider the behavior of the machine-language program when it is executed one instruction at a time. The first `irmovl` instruction would set register `%edx` to 10, the second would set the register to 3, and then the `rrmovl` instruction would read 3 from `%edx`. To imitate this behavior, our pipelined implementation should always give priority to the forwarding source in the earliest pipeline stage, since it holds the latest instruction in the program sequence setting the register. Thus, the logic in the HCL code above first tests the forwarding source in the execute stage, then those in the memory stage, and finally the sources in the write-back stage.

The forwarding priority between the two sources in either the memory or the write-back stages are only a concern for the instruction `popl %esp`, since only this instruction can write two registers simultaneously.

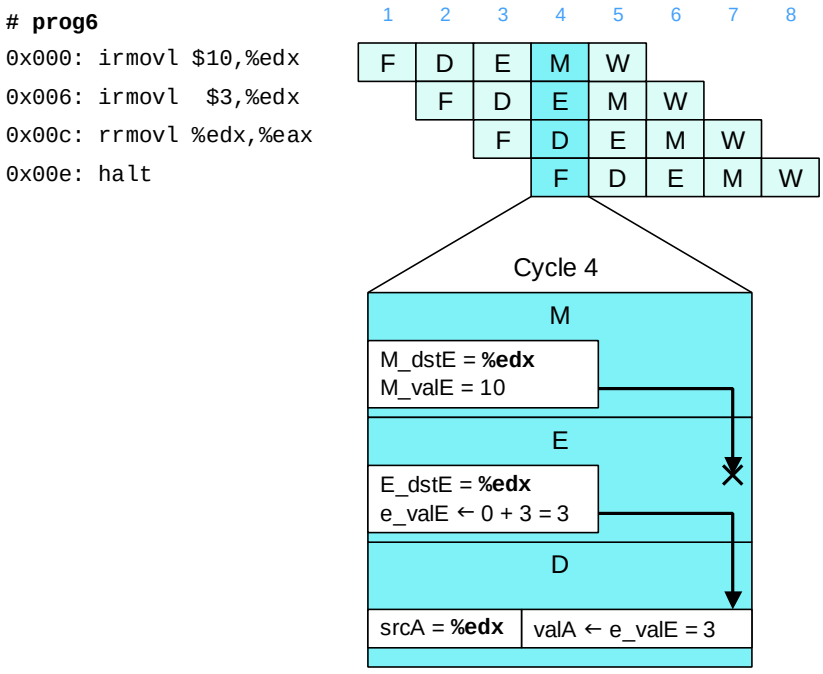


Figure 58: **Demonstration of forwarding priority.** In cycle 4, values for %edx are available from both the execute and memory stages. The forwarding logic should choose the one in the execute stage, since it represents the most recently generated value for this register.

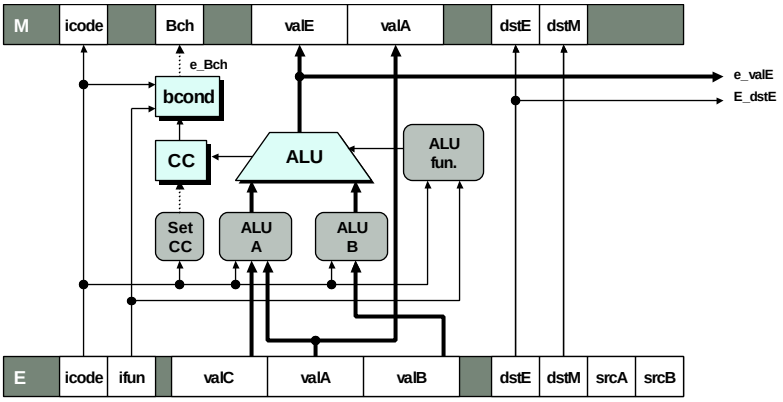


Figure 59: **PIPE execute stage logic.** This part of the design is very similar to the logic in the SEQ implementation.

Practice Problem 24:

Suppose the order of the third and fourth cases (the two forwarding sources from the memory stage) in the HCL code for new_E_valA were reversed. Describe the resulting behavior of the `rrmovl` instruction (line 5) for the following program:

```
1      irmovl $5, %edx
2      irmovl $0x100,%esp
3      rmmovl %edx,0(%esp)
4      popl %esp
5      rrmovl %esp,%eax
```

Practice Problem 25:

Suppose the order of the fifth and sixth cases (the two forwarding sources from the write-back stage) in the HCL code for new_E_valA were reversed. Write a Y86 program that would be executed incorrectly. Describe how the error would occur and its effect on the program behavior.

Practice Problem 26:

Write HCL code for the signal new_E_valB, giving the value for source operand valB supplied to pipeline register E.

Execute Stage

Figure 59 shows the execute stage logic for PIPE. The hardware units and the logic blocks are identical to those in SEQ, with an appropriate renaming of signals. We can see the signals e_valE and E_dstE directed toward the decode stage as one of the forwarding sources.

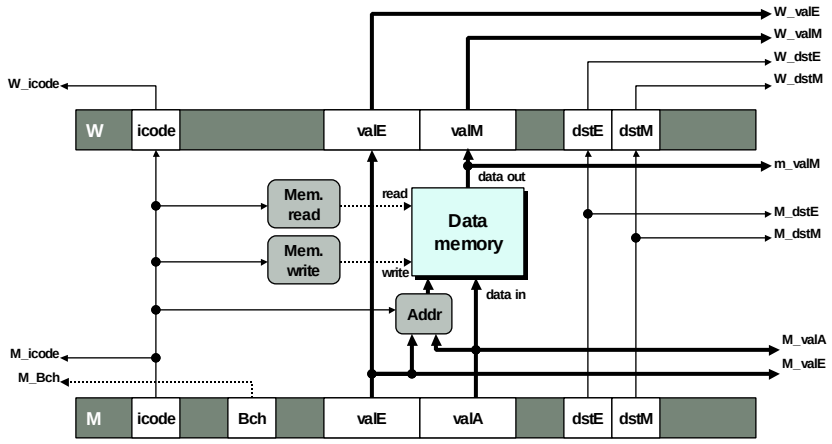


Figure 60: **PIPE memory stage logic.** Many of the signals from pipeline registers M and W are passed down to earlier stages to provide write-back results, instruction addresses, and forwarded results.

Memory Stage

Figure 60 shows the memory stage logic for PIPE. Comparing this to the memory stage for SEQ (Figure 28), we see that, as noted before, the block labeled "Data" in SEQ is not present in PIPE. This block served to select between data sources valP (for `call` instructions) and valA, but this selection is now performed by the block labeled "Sel+Fwd A" in the decode stage. All other blocks in this stage are identical to their counterparts in SEQ, with an appropriate renaming of the signals. In this figure you can also see that many of the values in pipeline registers and M and W are supplied to other parts of the circuit as part of the forwarding and pipeline control logic.

5.9 Pipeline Control Logic

We are now ready to complete our design for PIPE by creating the pipeline control logic. This logic must handle the following three control cases for which other mechanisms, such as data forwarding and branch prediction, do not suffice:

Processing `ret`: The pipeline must stall until the `ret` instruction reaches the write-back stage.

Load/use hazards: The pipeline must stall for one cycle between an instruction that reads a value from memory and an instruction that uses this value.

Mispredicted branches: By the time the branch logic detects that a jump should not have been taken, several instructions at the branch target will have started down the pipeline. These instructions must be removed from the pipeline.

We will go through the desired actions for each of these cases and then develop control logic to handle all of them.

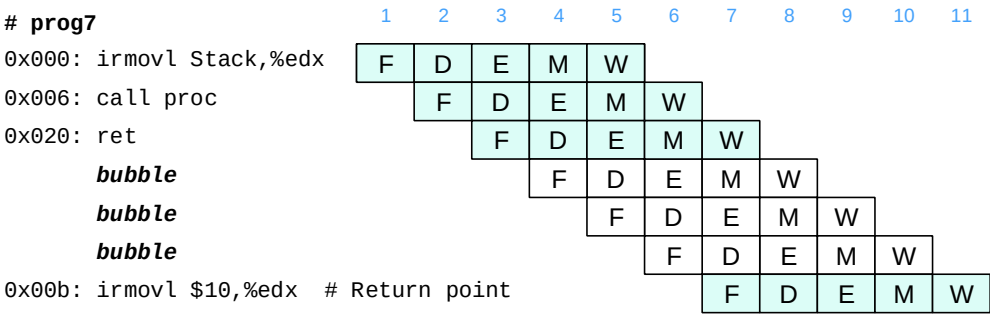


Figure 61: **Simplified view of `ret` instruction processing.** The pipeline should stall while the `ret` passes through the decode, execute, and memory stages, injecting three bubbles in the process. The PC selection logic will choose the return address as the instruction fetch address once the `ret` reaches the write-back stage (cycle 7).

Desired Handling of Special Control Cases

For the `ret` instruction, consider the following example program. This program is shown in assembly code, but with the addresses of the different instructions on the left for reference:

```
0x000:    irmovl Stack,%esp  # Intialize stack pointer
0x006:    call proc          # Procedure call
0x00b:    irmovl $10,%edx    # Return point
0x011:    halt
0x020: .pos 0x20
0x020: proc:
0x020:    ret               # proc:
0x021:    rrmovl %edx,%ebx    # Not executed
0x030: .pos 0x30
0x030: Stack:                # Stack: Stack pointer
```

Figure 61 shows how we want the pipeline to process the `ret` instruction. As with our earlier pipeline diagrams, this figure shows the pipeline activity with time growing to the right. Unlike before, the instructions are not listed in the same order they occur in the program, since this program involves a control flow where instruction are not executed in a linear sequence. Look at the instruction addresses to see from where the different instructions come in the program.

As this diagram shows, the `ret` instruction is fetched during cycle 3 and proceeds down the pipeline, reaching the write-back stage in cycle 7. While it passes through the decode, execute, and memory stages, the pipeline cannot do any useful activity. Instead, we want to inject three bubbles into the pipeline. Once the `ret` instruction reaches the write-back stage, the PC selection logic will set the program counter to the return address and therefore the fetch stage will fetch the `irmovl` instruction at the return point (address `0x00b`).

Figure 62 shows the actual processing of the `ret` instruction for the example program. The key observation here is that there is no way to inject a bubble into the fetch stage of our pipeline. On every cycle, the fetch stage reads some instruction from the instruction memory. Looking at the HCL code for implementing the

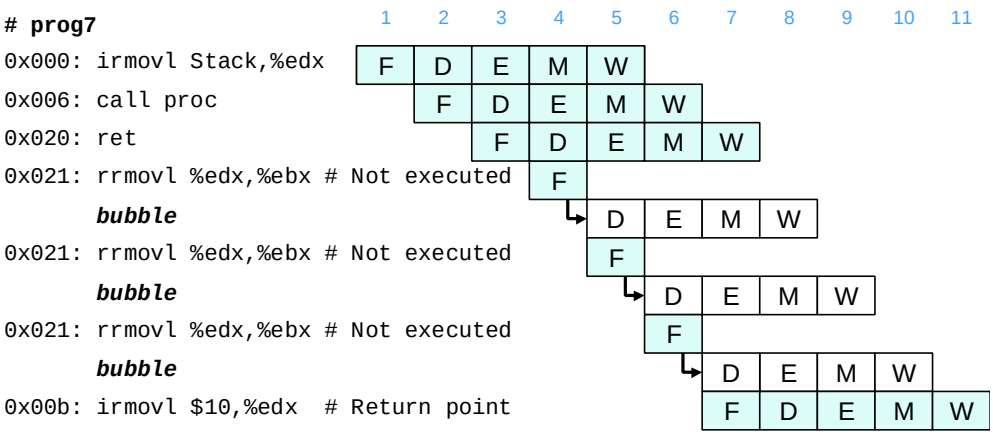


Figure 62: **Actual processing of the `ret` instruction.** The fetch stage repeatedly fetches the `rrmovl` instruction following the `ret` instruction, but then the pipeline control logic injects a bubble into the decode stage rather than allowing the `rrmovl` instruction to proceed. The resulting behavior is equivalent to that shown in Figure 61.

PC prediction logic in Section 5.8, we can see that for the `ret` instruction, the new value of the PC is predicted to be valP, the address of the following instruction. In our example program, this would be `0x021`, the address of the `rrmovl` instruction following the `ret`. This prediction is not correct for this example, nor would it be for most cases, but we are not attempting to predict return addresses correctly in our design. For three clock cycles, the fetch stage stalls, causing the `rrmovl` instruction to be fetched but then replaced by a bubble in the decode stage. This process is illustrated in Figure 62 by the three fetches, with an arrow leading down to the bubbles passing through the remaining pipeline stages. Finally, the `irmovl` instruction is fetched on cycle 7. Comparing Figure 62 with Figure 61, we see that our implementation achieves the desired effect, but with a slightly peculiar fetching of an incorrect instruction for three consecutive cycles.

For a load/use hazard, we have already described the desired pipeline operation in Section 5.7, as illustrated by the example of Figure 55. Only the `mrmovl` and `popl` instructions read data from memory. When either of these is in the execute stage, and an instruction requiring the destination register is in the decode stage, we want to hold back the second instruction in the decode stage and inject a bubble into the execute stage on the next cycle. After this, the forwarding logic will resolve the data hazard. The pipeline can hold back an instruction in the decode stage by keeping pipeline register D in a fixed state. In doing so, it should also keep pipeline register F in a fixed state, so that the next instruction will be fetched a second time. In summary, implementing this pipeline flow requires detecting the hazard condition, keeping pipeline register F and D fixed, and injecting a bubble into the execute stage.

To handle a mispredicted branch, consider the following program, shown in assembly code, but with the instruction addresses shown on the left for reference:

```
0x000:    xorl %eax,%eax
0x002:    jne  target        # Not taken
0x007:    irmovl $1, %eax    # Fall through
0x00d:    halt
0x00e: target:
```

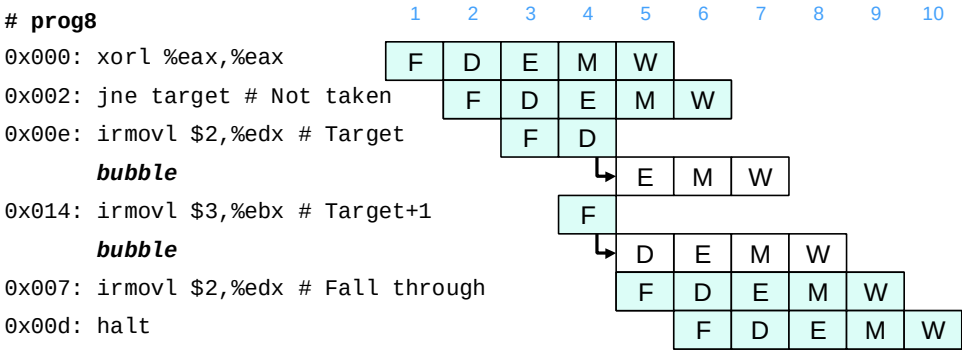


Figure 63: **Processing mispredicted branch instructions.** The pipeline predicts branches will be taken and so starts fetching instructions at the jump target. Two instructions are fetched before the misprediction is detected in cycle 4 when the jump instruction flows through the execute stage. In cycle 5, the pipeline *cancels* the two target instructions by injecting bubbles into the decode and execute stages, and it also fetches the instruction following the jump.

Condition	Trigger
Processing `ret`	IRET $\in$ {D_icode, E_icode, M_icode}
Load/use hazard	E_icode $\in$ {IMRMOVL, IPOPL} && E_dstM $\in$ {d_srcA, d_srcB}
Mispredicted branch	E_icode = IJXX && !e_Bch

Figure 64: **Detection conditions for pipeline control logic.** Three different conditions require altering the pipeline flow by either stalling the pipeline or by canceling partially executed instructions.

```
0x00e:    irmovl $2, %edx    # Target
0x014:    irmovl $3, %ebx    # Target+1
0x01a:    halt
```

Figure 63 shows how these instructions are processed. As before, the instructions are listed in the order they enter the pipeline, rather than the order they occur in the program. Since the jump instruction is predicted as being taken, the instruction at the jump target will be fetched in cycle 3, and the instruction following this one will be fetched in cycle 4. By the time the branch logic detects that the jump should not be taken at during cycle 4, two instructions have been fetched that should not continue being executed. Fortunately, neither of these instructions has caused a change in the programmer-visible state. That can only occur when an instruction reaches the execute stage, where it can cause the condition codes to change. We can simply *cancel* (sometimes called *instruction squashing*) the two misfetched instructions by injecting bubbles into the decode and execute instructions on the following cycle while also fetching the instruction following the jump instruction. The two misfetched instructions will then simply disappear from the pipeline.

Detecting Special Control Conditions

Figure 64 summarizes the conditions requiring special pipeline control. It gives HCL expressions describing the conditions under which the three special cases arise. These expressions are implemented by simple

blocks of combinational logic that must generate their results before the end of the clock cycle in order to control the action of the pipeline registers as the clock rises to start the next cycle. During a clock cycle, pipeline register D, E, and M hold the states of the instructions that are in the decode, execute, and memory pipeline stages, respectively. As we approach the end of the clock cycle, signals d_srcA an d_srcB will be set to the register IDs of the source operands for the instruction in the decode stage. Detecting a `ret` instruction as it passes through the pipeline simply involves checking the instruction codes of the instructions in the decode, execute, and memory stages. Detecting a load/use hazard involves checking the instruction type (`mrmovl` or `popl`) of the instruction in the execute stage, and comparing its destination register with the source registers of the instruction in the decode stage. The pipeline control logic should detect a mispredicted branch while the jump instruction is in the execute stage, so that it can set up the conditions required to recover from the misprediction as the instruction enters the memory stage. When a jump instruction is in the execute stage, the signal e_Bch indicates whether or not the jump should be taken.

Pipeline Control Mechanisms

Figure 65 shows low-level mechanisms that allow the pipeline control logic to hold back an instruction in a pipeline register or to inject a bubble into the pipeline. These mechanisms involve small extensions to the basic clocked register described in Section 2.5. Suppose that each pipeline register has two control inputs stall and bubble. The settings of these signals determine how the pipeline register is updated as the clock rises. Under normal operation (A), both of these inputs are set to 0, causing the register to load its input as its new state. When the stall signal is set to 1 (B), the updating of the state is disabled. Instead, the register will remain in its previous state. This makes it possible to hold back an instruction in some pipeline stage. When the bubble signal is set to 1 (C), the state of the register will be set to some fixed *reset configuration* giving a state equivalent to that of a `nop` instruction. The particular pattern of 1s and 0s for a pipeline register's reset configuration depends on the set of fields in the pipeline register. For example, to inject a bubble into pipeline register D, we want the icode field to be set to the constant value `INOP` (Figure 24). To inject a bubble into pipeline register E, we want the icode field to be set to `INOP` and the dstE, dstM, srcA, and srcB fields to be set to the constant `RNONE`. Determining the reset configuration is one of the tasks for the hardware designer in designing a pipeline register. We will not concern ourselves with the details here. We will consider it an error to set both the bubble and the stall signals to 1.

The table in Figure 66 shows the actions the different pipeline stages should take for each of the three special conditions. Each involves some combination of normal, stall, and bubble operations for the pipeline registers.

In terms of timing, the stall and bubble control signals for the pipeline registers are generated by blocks of combinational logic. These values must be valid as the clock rises, causing each of the pipeline registers to either load, stall, or bubble as the next clock cycle begins. With this small extension to the pipeline register designs, we can implement a complete pipeline, including all of its control, using the basic building blocks of combinational logic, clocked registers, and random-access memories.

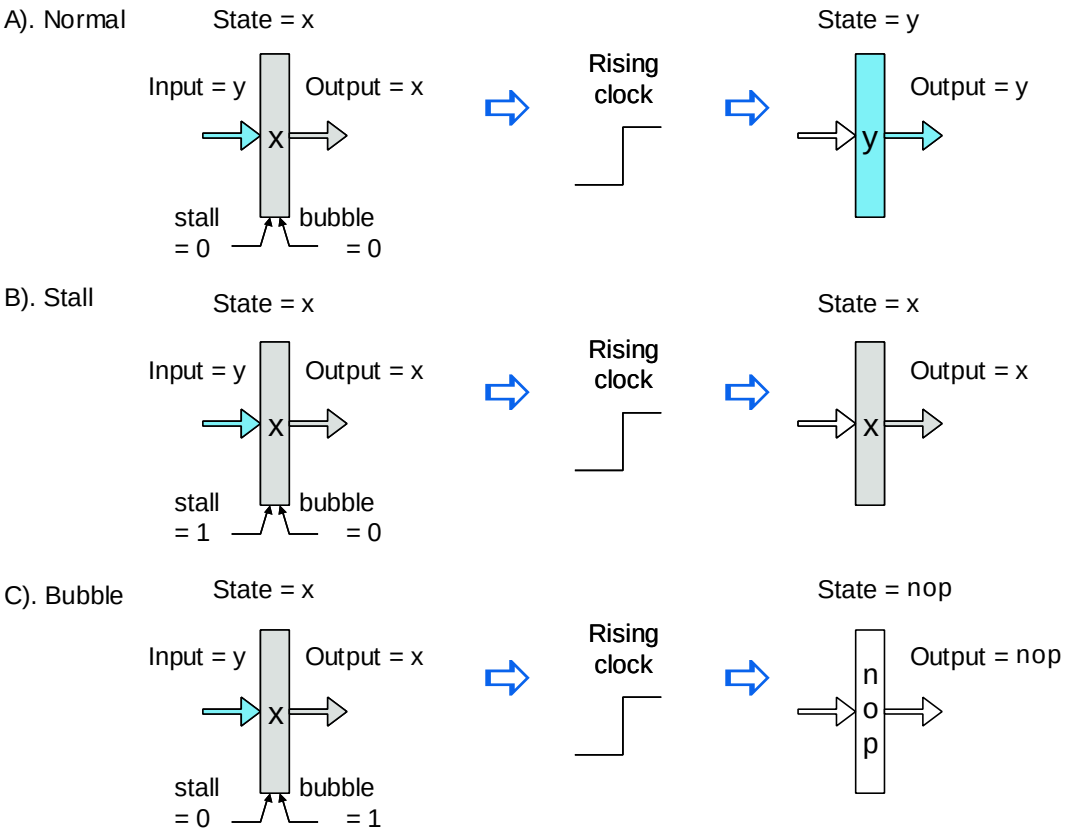


Figure 65: **Additional pipeline register operations.** Under normal conditions (A), the state and output of the register are set to the value at the input when the clock rises. When operated in *stall* mode (B), the state is held fixed at its previous value. When operated in *bubble* mode (C), the state is overwritten with that of a `nop` operation.

	Pipeline register				
Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal

Figure 66: **Actions for pipeline control logic.** The different conditions require altering the pipeline flow by either stalling the pipeline or by canceling partially executed instructions.

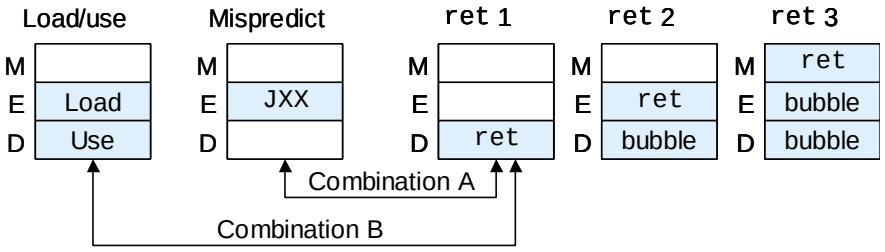


Figure 67: **Pipeline states for special control conditions.** The two pairs indicated can arise simultaneously.

Combinations of Control Conditions

In our discussion of the special pipeline control conditions so far, we assumed that at most one special case could arise during any single clock cycle. A common bug in designing a system is to fail to handle instances where multiple special conditions arise simultaneously. Let's analyze such possibilities. Figure 67 diagrams the pipeline states that cause the special control conditions. These diagrams show blocks for the decode, execute, and memory stages. The shaded boxes represent particular constraints that must be satisfied for the condition to arise. A load/use hazard requires that the instruction in the execute stage reads a value from memory into a register, and that the instruction in the decode stage has this register as a source operand. A mispredicted branch requires the instruction in the execute stage to have a jump instruction. There are three possible cases for `ret`—the instruction can be in either the decode, execute, or memory stage. As the `ret` instruction moves through the pipeline, the earlier pipeline stages will have bubbles.

We can see by these diagrams that most of the control conditions are mutually exclusive. For example, it is not possible to have a load/use hazard and a mispredicted branch simultaneously, since one requires a load instruction (`mrmovl` or `popl`) in the execute stage, while the other requires a jump. Similarly, the second and third `ret` combinations cannot occur at the same time as a load/use hazard or a mispredicted branch. Only the two combinations indicated by arrows can arise simultaneously.

Combination A involves a not-taken jump instruction in the execute stage and a `ret` instruction in the decode stage. Setting up this combination requires the `ret` to be at the target of a not-taken branch. The pipeline control logic should detect that the branch was mispredicted and therefore cancel the `ret` instruction.

> **Practice Problem 27**:
>
> Write a Y86 assembly language program that causes combination A to arise and determines whether the control logic handles it correctly.

Combining the control actions for the combination A conditions (Figure 66), we get the following pipeline control actions (assuming that either a bubble or a stall overrides the normal case):

	Pipeline register				
Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal
Combination	stall	bubble	bubble	normal	normal

That is, it would be handled like a mispredicted branch, but with a stall in the fetch stage. Fortunately, on the next cycle, the PC selection logic will choose the address of the instruction following the jump, rather than the predicted program counter, and so it does not matter what happens with the pipeline register F. We conclude that the pipeline will correctly handle this combination.

Combination B involves a load/use hazard, where the loading instruction sets register `%esp`, and the `ret` instruction then uses this register as a source operand, since it must pop the return address from the stack. The pipeline control logic should hold back the `ret` instruction in the decode stage.

Practice Problem 28:

Write a Y86 assembly language program that causes combination B to arise and completes with a `halt` instruction if the pipeline operates correctly.

Combining the control actions for the combination B conditions (Figure 66), we get the following pipeline control actions:

	Pipeline register				
Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Combination	stall	bubble+stall	bubble	normal	normal
Desired	stall	stall	bubble	normal	normal

If both sets of actions were triggered, the control logic would try to stall the `ret` instruction to avoid the load/use hazard but also inject a bubble into the decode stage due to the `ret` instruction. Clearly, we do not want to the pipeline to perform both sets of actions. Instead, we want it just take the actions for the load/use hazard. The actions for processing the `ret` instruction should be delayed for one cycle.

This analysis shows that combination B requires special handling. In fact, our original implementation of the PIPE control logic did not handle this combination correctly. Even though the design had passed many simulation tests, it had a subtle bug that was uncovered only by the analysis we have just shown. When a program having combination B was executed, the control logic would set both the bubble and the stall signals for pipeline register D to 1. This example shows the importance of systematic analysis. It would be unlikely to uncover this bug by just running normal programs. If left undetected, the pipeline would not faithfully implement match the ISA behavior.

Control Logic Implementation

Figure 68 shows the overall structure of the pipeline control logic. Based on signals from the pipeline registers and pipeline stages, the control logic generates stall and bubble control signals for the pipeline

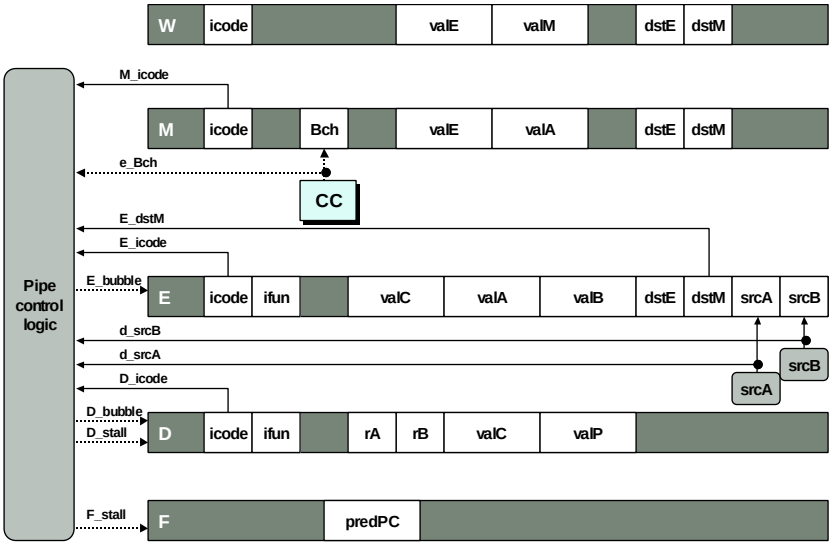


Figure 68: **PIPE pipeline control logic.** This logic overrides the normal flow of instructions through the pipeline to handle special conditions such as procedure returns, mispredicted branches, and load/use hazards.

registers. We can combine the detection conditions of Figure 64 with the actions of Figure 66 to create HCL descriptions for the different pipeline control signals.

Pipeline register F must be stalled for either a load/use hazard or a `ret` instruction:

```
bool F_stall =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVL, IPOPL } &&
         E_dstM in { d_srcA, d_srcB } ||
        # Stalling at fetch while ret passes through pipeline
        IRET in { D_icode, E_icode, M_icode };
```

Practice Problem 29:

Write HCL code for the signal D_stall in the PIPE implementation.

Pipeline register D must be set to bubble for a mispredicted branch or a `ret` instruction. As the analysis in the preceding section shows, however, it should not inject a bubble when there is a load/use hazard in combination with a `ret` instruction:

```
bool D_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Bch) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        !(E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }) &&
          IRET in { D_icode, E_icode, M_icode };
```

Practice Problem 30:

Write HCL code for the signal E_bubble in the PIPE implementation.

This covers all of the special pipeline control signal values. In the complete HCL code for PIPE, all other pipeline control signals are set to 0.

5.10 Performance Analysis

We can see that the conditions requiring special action by the pipeline control logic all cause our pipeline to fall short of the goal of issuing a new instruction on every clock cycle. We can measure this inefficiency by determining how often a bubble gets injected into the pipeline, since these cause unused pipeline cycles. A return instruction generates three bubbles, a load/use hazard generates one, and a mispredicted branch generates two. We can quantify the effect these penalties have on the overall performance by computing an estimate of the average number of clock cycles PIPE would require per instruction it executes, a measure known as the CPI (for "cycles per instruction"). This measure is the reciprocal of the average throughput of the pipeline, but with time measured in clock cycles rather than picoseconds. It is a useful measure of the architectural efficiency of a design.

Another way to think about CPI is to imagine we run the processor on some benchmark program and observe the operation of the execute stage. On each cycle, the execute stage would either process an instruction, and this instruction would then continue through the remaining stages to completion, or it would process a bubble, injected due to one of the three special cases. If the stage processes a total of C_i instructions and C_b bubbles, then the processor has required around $C_i + C_b$ total clock cycles to execute C_i instructions. We say "around" because we ignore the cycles required to start the instructions flowing through the pipeline. We can then compute the CPI for this benchmark as follows:

$$CPI \quad = \quad \frac{C_i + C_b}{C_i} \quad = \quad 1.0 + \frac{C_b}{C_i}$$

That is, the CPI equals 1.0 plus a penalty term C_b/C_i indicating the average number of bubbles injected per instruction executed. Since only three different instruction types can cause a bubble to be injected, we can break this penalty term into three components:

$$CPI \quad = \quad 1.0 + lp + mp + rp$$

where lp (for "load penalty") is the average frequency with which bubbles are injected while stalling for load/use hazards, mp (for "mispredicted branch penalty") is the average frequency with which bubbles are injected when canceling instructions due to mispredicted branches, and rp (for "return penalty") is the average frequency with which bubbles are injected while stalling for `ret` instructions. Each of these penalties indicates the total number of bubbles injected for the stated reason (some portion of C_b) divided by the total number of instructions that were executed (C_i).

To estimate each of these penalties, we need to know how frequently the relevant instructions (load, conditional branch, and return) occur, and for each of these how frequently the particular condition arises. Let's pick the following set of frequencies for our CPI computation (these are comparable to measurements reported in [3] and [4]):

- Load instructions (`mrmovl` and `popl`) account for 25% of all instructions executed. Of these, 20% cause load/use hazards.
- Conditional branches account for 20% of all instructions executed. Of these, 60% are taken and 40% are not taken.
- Return instructions account for 2% of all instructions executed.

We can therefore estimate each of our penalties as the product of the frequency of the instruction type, the frequency the condition arises, and the number of bubbles that get injected when the condition occurs:

Cause	Name	Instruction frequency	Condition frequency	Bubbles	Product
Load/Use	lp	0.25	0.20	1	0.05
Mispredict	mp	0.20	0.40	2	0.16
Return	rp	0.02	1.00	3	0.06
Total Penalty					0.27

The sum of the three penalties is 0.27, giving a CPI of 1.27.

Our goal was to design a pipeline that can issue one instruction per cycle, giving a CPI of 1.0. We did not quite meet this goal, but the overall performance is still quite good. We can also see that any effort to reduce the CPI further should focus on mispredicted branches. They account of 0.16 of our total penalty of 0.27, because conditional branches are common, our prediction strategy often fails, and we cancel two instructions for every misprediction.

Practice Problem 31:

Suppose we use a branch prediction strategy that achieves a success rate of 65%, such as backward taken, forward not-taken, as described in Section 5.3. What would be the impact on CPI, assuming all of the other frequencies are not affected?

5.11 Unfinished Business

We have created a structure for the PIPE pipelined microprocessor, designed the control logic blocks, and implemented pipeline control logic to handle special cases where normal pipeline flow does not suffice. Still, PIPE lacks several key features that would be required in an actual microprocessor design. We highlight a few of these and discuss what would be required to add them.

Exception Handling

When a machine-level program encounters an error condition, such as an invalid instruction code or an out-of-range instruction or data address, it causes a break in the program flow, called an *exception*. An exception behaves like a procedure call, invoking an *exception handler*, a procedure that is part of the operating system. We describe more about exception handling in Chapter **??**. Executing the `halt` instruction should also

trigger an exception. Exception handling is part of the instruction set architecture for a processor. Generally, the exception should cause the processing to stop at a point either just before or just after the instruction that causes the exception, depending on the exception type. That is, it should appear that all instructions up to the exception point have completed, but none of the instructions following this point should have had any affect on the programmer-visible state.

In a pipelined system, exception handling involves several subtleties. First it is possible to have exceptions triggered by multiple instructions simultaneously. For example, during one cycle of pipeline operation, we could have the instruction memory report an out-of-bounds instruction address for the instruction in the fetch stage, the data memory report an out-of-bounds data address for the instruction in the memory stage, and the control logic report an invalid code for the instruction in the decode stage. We must determine which of these exceptions the processor should report to the operating system. The basic rule is to put priority on the exception triggered by the instruction that is furthest along the pipeline. In the example above, this would be the out-of-bounds address attempted by the instruction in the memory stage. In terms of the machine-language program, the instruction in the memory stage should appear to execute before those in the decode or fetch stage begin, and therefore only this exception should be reported to the operating system.

A second subtlety occurs when an instruction is first fetched and begins execution, then causes an exception, and later is canceled due to a mispredicted branch. The following is an example of such a program in its object code form:

```
0x000: 6300         |    xorl %eax,%eax
0x002: 740e000000   |    jne  Target      # Not taken
0x007: 308001000000 |    irmovl $1, %eax  # Fall through
0x00d: 10           |    halt
0x00e:              | Target:
0x00e: ff           |    .byte 0xFF       # Invalid instruction code
```

In this program, the pipeline will predict that the branch should be taken, and so it will fetch and attempt to use a byte with value `0xFF` as an instruction (generated in the assembly code using the `.byte` directive). The decode stage will therefore detect an invalid instruction exception. Later, the pipeline will discover that the branch should not be taken, and so the instruction at address `0x00e` should never even have been fetched. The pipeline control logic will cancel this instruction, but we want to avoid raising an exception.

A third subtlety arises because a pipelined processor updates different parts of the system state in different stages. It is possible for an instruction following one causing an exception to alter some part of the state before the excepting instruction completes. For example, consider the following code sequence, in which we assume that user programs are not allowed to access addresses greater than `0xc0000000` (as is the case for current versions of Linux, as is discussed in Chapter **??**):

```
1     irmovl $0,%esp        # Set stack pointer to 0
2     pushl %eax            # Attempts to write to 0xfffffffc
3     addl  %ecx,%eax       # Sets condition codes
```

The `pushl` instruction causes an address exception, because decrementing the stack pointer causes it to wrap around to `0xfffffffc`. This exception is detected in the memory stage. On the same cycle, the `addl` instruction is in the execute stage, and it will cause the condition codes to be set to new values. This

would violate our requirement that none of the instructions following the exception point should have had any effect on the system state.

In general, we can both correctly choose among the different exceptions and avoid raising exceptions for instructions that are fetched due to mispredicted branches by merging the exception-handling logic into the pipeline structure. We add a special field `exc` to every pipeline register, giving the exception status for the instruction at that pipeline stage. If an instruction generates an exception at some stage in its processing, the status field is set to indicate the nature of the exception. The exception status propagates through the pipeline with the rest of the information for that instruction, until it reaches the write-back stage. At this point, pipeline control logic detects the occurrence of the exception and initiates the fetching of the code for the exception handler.

To avoid having any updating of the programmer-visible state by instructions beyond the exception point, the pipeline control logic should be modified to disable any updating of the condition code register or the data memory when an instruction in the memory or write-back stages has caused an exception. In the example program above, the control logic would detect that the `pushl` in the memory stage has caused an exception, and therefore the updating of the condition code register by the `addl` instruction would be disabled. (In the simulator for PIPE that accompanies this text, you will see an implementation of these techniques for handling exceptions in a pipelined processor.)

Let's consider how this method of handling exceptions deals with the subtleties we have mentioned. When an exception occurs in one or more stages of a pipeline, the information is simply stored in the exception status fields of the pipeline registers. The event has no effect on the flow of instructions in the pipeline until an excepting instruction reaches the final pipeline stage, except to disable any updating of the programmer-visible state (the condition code register or the memory) by later instructions in the pipeline. Since instructions reach the write-back stage in the same order as they would be executed in a nonpipelined processor, we are guaranteed that the first instruction encountering an exception will be the first one to trigger the transfer of control to the exception handler. If some instruction is fetched but later canceled, any exception status information about the instruction gets canceled as well. No instruction following one that causes an exception can alter the programmer-visible state. The simple rule of carrying the exception status together with all other information about an instruction through the pipeline provides a simple and reliable mechanism for handling exceptions.

Multicycle Instructions

All of the instructions in the Y86 instruction set involve simple operations such as adding numbers. These can be processed in a single clock cycle within the execute stage. In a more complete instruction set, we would also need to implement instructions requiring more complex operations, such as integer multiplication and division, and floating-point operations. In a medium-performance processor such as PIPE, typical execution times for these operations range from 3 or 4 cycles for floating-point addition up to 32 for integer division. To implement these instructions, we require both additional hardware to perform the computations, and a mechanism to coordinate the processing of these instructions with the rest of the pipeline.

One simple approach to implementing multicycle instructions is to simply expand the capabilities of the execute stage logic with integer and floating-point arithmetic units. An instruction remains in the execute stage for as many clock cycles as it requires, causing the fetch and decode stages to stall. This approach is

simple to implement, but the resulting performance is not very good.

Better performance can be achieved by handling the more complex operations with special hardware functional units that operate independently of the main pipeline. Typically, there is one functional unit for performing integer multiplication and division, and another for performing floating-point operations. As an instruction enters the decode stage, it can be *issued* to the special unit. While the unit performs the operation, the pipeline continues processing other instructions. Typically, the floating-point unit is itself pipelined, and thus multiple operations can execute concurrently in the main pipeline and in the different units.

The operations of the different units must be synchronized to avoid incorrect behavior. For example, if there are data dependencies between the different operations being handled by different units, the control logic may need to stall one part of the system until the results from an operation handled by some other part of the system have been completed. Often, different forms of forwarding are used to convey results from one part of the system to other parts, just as we saw between the different stages of PIPE. The overall design becomes more complex than we have seen with PIPE, but the same techniques of stalling, forwarding, and pipeline control can be used to make the overall behavior match the sequential ISA model.

Interfacing with the Memory System

In our presentation of PIPE, we assumed that both the instruction fetch unit and the data memory could read or write any memory location in one clock cycle. We also ignored the possible hazards caused by self-modifying code where one instruction writes to the region of memory from which later instructions are fetched. Furthermore, we reference memory locations according to their virtual addresses, and these require a translation into physical addresses before the actual read or write operation can be performed. Clearly, it is unrealistic to do all of this processing in a single clock cycle. Even worse, the memory values being accessed may reside on disk, requiring millions of clock cycles to read into the processor memory.

As will be discussed in Chapters **??** and **??**, the memory system of a processor uses a combination of multiple hardware memories and operating system software to manage the virtual memory system. The memory system is organized as a hierarchy, with faster, but smaller memories holding a subset of the memory being backed up by slower and larger memories. At the level closest to the processor, the *cache* memories provide fast access to the most heavily referenced memory locations. A typical processor has two first-level caches—one for reading instructions and one for reading and writing data. Another type of cache memory, known as a *translation look-aside buffer*, or TLB, provides a fast translation from virtual to physical addresses. Using a combination of TLBs and caches, it is indeed possible to read instructions and read or write data in a single clock cycle most of the time. Thus, our simplified view of memory referencing by our processors is actually quite reasonable.

Although the caches hold the most heavily referenced memory locations, there will be times when a cache *miss* occurs, where some reference is made to a location that is not held in the cache. In the best case, the missing data can be retrieved from a higher level cache or from the main memory of the processor, requiring 3 to 20 clock cycles. Meanwhile, the pipeline simply stalls, holding the instruction in the fetch or memory stage until the cache can perform the read or write operation. In terms of our pipeline design, this can be implemented by adding more stall conditions to the pipeline control logic. A cache miss and the consequent synchronization with the pipeline is handled completely by hardware, keeping the time required down to a small number of clock cycles.

In some cases, the memory location being referenced is actually stored in the disk memory. When this occurs, the hardware signals a *page fault* exception. Like other exceptions, this will cause the processor to invoke the operating system's exception handler code. This code will then set up a transfer from the disk to the main memory. Once this completes, the operating system will return back to the original program, where the instruction causing the page fault will be reexecuted. This time the memory reference will succeed, although it might cause a cache miss. Having the hardware invoke an operating system routine, which then returns control back to the hardware, allows the hardware and system software to cooperate in the handling of page faults. Since accessing a disk can require millions of clock cycles, the several thousand cycles of processing performed by the OS page fault handler has little impact on performance.

From the perspective of the processor, the combination of stalling to handle short duration cache misses and exception handling to handle long duration page faults takes care of any unpredictability in memory access times due to the structure of the memory hierarchy.

Aside: State-of-the-Art Microprocessor Design
A five-stage pipeline, such as we have shown with the PIPE processor, represented the state of the art in processor design in the mid-1980s. The prototype RISC processor developed by Patterson's research group at Berkeley formed the basis for the first SPARC processor, developed by Sun Microsystems in 1987. The processor developed by Hennessy's research group at Stanford was commercialized by MIPS Technologies (a company founded by Hennessy) in 1986. Both of these used five-stage pipelines. The Intel i486 processor also uses a five-stage pipeline, although with a different partitioning of responsibilities among the stages, with two decode stages and a combined execute/memory stage [2].

These pipelined designs are limited to a throughput of at most one instruction per clock cycle. The CPI (for "cycles per instruction") measure described in Section 5.10 can never be less than 1.0. The different stages can only process one instruction at a time. More recent processors support *superscalar* operation, meaning that they can achieve a CPI less than 1.0 by fetching, decoding, and executing multiple instructions in parallel. As superscalar processors have become widespread, the accepted performance measure has shifted from CPI to its reciprocal—the average number of instructions executed per cycle, or IPC. It can exceed 1.0 for superscalar processors. The most advanced designs use a technique known as *out-of-order* execution to execute multiple instructions in parallel, possibly in a totally different order than they occur in the program, while preserving the overall behavior implied by the sequential ISA model. This form of execution is described in Chapter **??** as part of our discussion of program optimization.

Pipelined processors are not just historical artifacts, however. The majority of processors sold are used in embedded systems, controlling automotive functions, consumer products, and other places where the processor is not directly visible to the system user. In these applications, the simplicity of a pipelined processor, such as the one we have explored in this chapter, reduces its cost and power requirements compared to higher performance models. **End Aside.**

6 Summary

We have seen that the instruction set architecture, or ISA, provides a layer of abstraction between the behavior of a processor—in terms of the set of instructions and their encodings—and how the processor is implemented. The ISA provides a very sequential view of program execution, with one instruction executed to completion before the next one begins.

We defined the Y86 instruction set by starting with the IA32 instructions, and simplifying the data types, address modes, and instruction encoding considerably. The resulting ISA has attributes of both RISC and CISC instruction sets. We then organized the processing required for the different instructions into a series of six stages, where the operations at each stage vary according to the instruction being executed. From this,

we constructed the SEQ processor, in which each clock cycle consists of stepping an instruction through each of the six stages. By reordering the stages, we created the SEQ+ design, where the first stage selects the value of the program counter to be used for fetching the current instruction.

Pipelining improves the throughput performance of a system by letting the different stages operate concurrently. At any given time, multiple operations are being processed by the different stages. In introducing this concurrency, we must be careful to provide the user-visible, program-level behavior as would a sequential execution of the program. We introduced pipelining by adding pipeline registers to SEQ+ and rearranging cycles to create the PIPE– pipeline. We then enhanced the pipeline performance by adding forwarding logic to speed the sending of a result from one instruction to another. Several special cases require additional pipeline control logic to stall or cancel some of the pipeline stages.

In this chapter, we have learned several important lessons about processor design:

- *Managing complexity is a top priority*. We want to make optimum use of the hardware resources to get maximum performance at minimum cost. We did this by creating a very simple and uniform framework for processing all of the different instruction types. With this framework, we could share the hardware units among the logic for processing the different instruction types.
- *We do not need to implement the ISA directly*. A direct implementation of the ISA would imply a very sequential design. To achieve higher performance we want to exploit the ability in hardware to perform many operations simultaneously. This led to the use of a pipelined design. By careful design and analysis, we can handle the various pipeline hazards, so that the overall effect of running a program exactly matches what would be obtained with the ISA model.
- *Hardware designers must be meticulous.* Once a chip has been fabricated, it is nearly impossible to correct any errors. It is very important to get the design right on the first try. This means carefully analyzing different instruction types and combinations, even ones that do not seem to make sense, such as popping to the stack pointer. Designs must be thoroughly tested with systematic simulation test programs. In developing the control logic for PIPE, our design had a subtle bug that was uncovered only after a careful and systematic analysis of control combinations.

6.1 Y86 Simulators

The lab materials for this chapter include simulators for the SEQ, SEQ+, and PIPE processors. Each simulator has two versions:

- The GUI (graphic user interface) version displays the memory, program code, and processor state in graphic windows. This provides a way to readily see how the instructions flow through the processors. The control panel also allows you to reset, single-step, or run the simulator interactively. These versions require the Tcl scripting language and the Tk graphic library.
- The text version runs the same simulator, but it only displays information by printing to the terminal. This version is not as useful for debugging, but it allows automated testing of the processor, and it can be run on systems that do not support Tcl/Tk.

The control logic for the simulators is generated by translating the HCL declarations of the logic blocks into C code. This code is then compiled and linked with the rest of the simulation code. This combination makes it possible for you to test out variants of the original designs using the simulators. Testing scripts are also available that thoroughly exercise the different instructions and the different hazard possibilities.

Bibliographic Notes

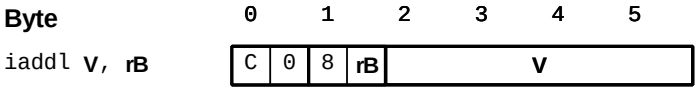
For those interested in learning more about logic design, Katz's logic design textbook [5] is a standard introductory text, emphasizing the use of hardware description languages.

Hennessy and Patterson's computer architecture textbook [4] provides extensive coverage of processor design, including both simple pipelines such as the one we have presented here, and more advanced processors that execute more instructions in parallel. Shriver and Smith [6] give a very thorough presentation of an Intel-compatible IA32 processor manufactured by AMD.

Homework Problems

Homework Problem 32 ♦:

In our example Y86 programs, such as the `Sum` function shown in Figure 5, we encounter many cases (e.g., lines 12 and 13 and lines 14 and 15) in which we want to add a constant value to a register. This requires first using an `irmovl` instruction to set a register to the constant, and then an `addl` instruction to add this value to the destination register. Suppose we want to add a new instruction `iaddl` with the following format:



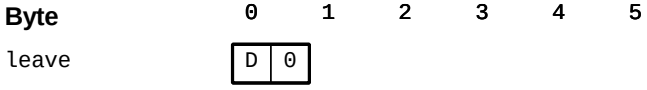
This instruction adds the constant value `V` to register `rB`. Describe the computations performed to implement this instruction. Use the computations for `irmovl` and `OPl` (Figure 16) as a guide.

Homework Problem 33 ♦:

As described in Section **??**, the IA32 instruction `leave` can be used to prepare the stack for returning. It is equivalent to the following Y86 code sequence:

```
1   rrmovl %ebp, %esp   Set stack pointer to beginning of frame
2   popl   %ebp         Restore saved %ebp and set stack ptr to end of caller's frame
```

Suppose we add this instruction to the Y86 instruction set, using the following encoding:



Describe the computations performed to implement this instruction. Use the computations for `popl` (Figure 18) as a guide.

Homework Problem 34 ♦♦:

The file `seq-full.hcl` contains the HCL description for SEQ, along with the declaration of a constant `IIADDL` having hexadecimal value `C`, the instruction code for `iaddl`. Modify the HCL descriptions of the control logic blocks to implement the `iaddl` instruction, as described in homework problem 32. See the lab material for directions on how to generate a simulator for your solution and how to test it.

Homework Problem 35 ♦♦:

The file `seq-full.hcl` also contains the declaration of a constant `ILEAVE` having hexadecimal value `D`, the instruction code for `leave`, as well as the declaration of a constant `REBP` having value `7`, the register ID for `%ebp`. Modify the HCL descriptions of the control logic blocks to implement the `leave` instruction, as described in homework problem 33. See the lab material for directions on how to generate a simulator for your solution and how to test it.

Homework Problem 36 ♦♦♦:

Suppose we wanted to create a lower-cost pipelined processor based on the structure we devised for PIPE– (Figures 39 and 41), without any bypassing. This design would handle all data dependencies by stalling until the instruction generating a needed value has passed through the write-back stage.

The file `pipe-stall.hcl` contains modified version of the HCL code for PIPE in which the bypassing logic has been disabled. That is, the signals `e_valA` and `e_valB` are simply declared as follows:

```
##  DO NOT MODIFY THE FOLLOWING CODE.
## No forwarding.  valA is either valP or value from register file
int new_E_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        1 : d_rvalA;  # Use value read from register file
];

## No forwarding.  valB is value from register file
int new_E_valB = d_rvalB;
```

Modify the pipeline control logic at the end of this file so that it correctly handles all possible control and data hazards. As part of your design effort, you should analyze the different combinations of control cases, as we did in the design of the pipeline control logic for PIPE. You will find that many different combinations can occur, since many more conditions require the pipeline to stall. Make sure your control logic handles each combination correctly. See the lab material for directions on how to generate a simulator for your solution and how to test it.

Homework Problem 37 ♦♦:

The file `pipe-full.hcl` contains a copy of the PIPE HCL description, along with a declaration of the constant value `IIADDL`. Modify this file to implement the `iaddl` instruction, as described in homework problem 32. See the lab material for directions on how to generate a simulator for your solution and how to test it.

Homework Problem 38 ♦♦♦:

The file `pipe-full.hcl` also contains declarations of constants `ILEAVE` and `REBP`. Modify this file to implement the `leave` instruction, as described in homework problem 33. See the lab material for directions on how to generate a simulator for your solution and how to test it.

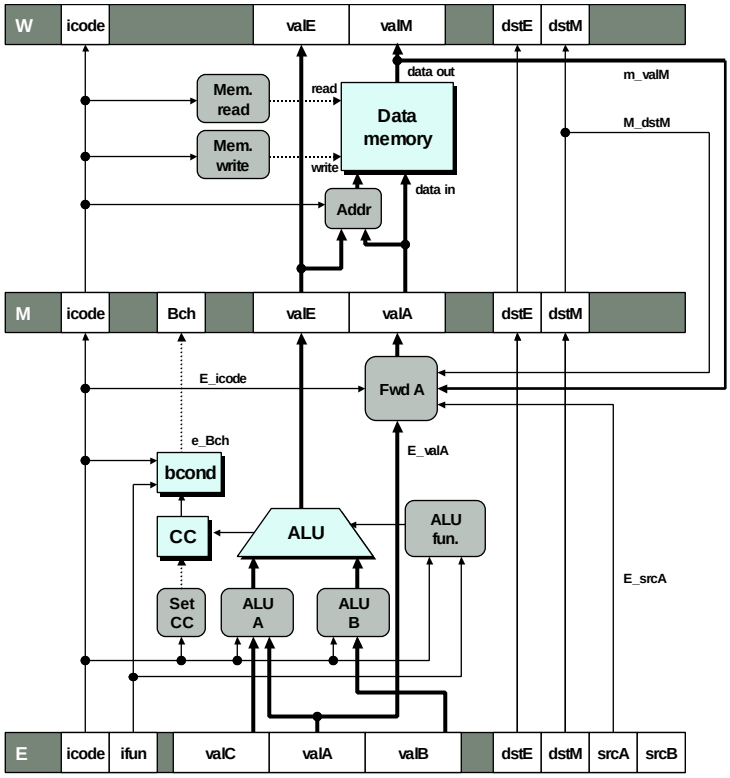


Figure 69: **Execute and memory stages capable of load forwarding.** By adding a bypass path from the memory output to the source of valA in pipeline register M, we can use forwarding rather than stalling for one form of load/use hazard. This is the subject of homework problem 41.

Homework Problem 39 ♦♦♦:

The file `pipe-nt` contains a copy of the HCL code for PIPE, plus a declaration of the constant `J_YES` with value `0`, the function code for an unconditional jump instruction. Modify the branch prediction logic so that it predicts conditional jumps as being not-taken while continuing to predict unconditional jumps and `call` as being taken. You will need to devise a way to get valC, the jump target address, to pipeline register M to recover from mispredicted branches. See the lab material for directions on how to generate a simulator for your solution and how to test it.

Homework Problem 40 ♦♦♦:

The file `pipe-btfnt` contains a copy of the HCL code for PIPE, plus a declaration of the constant `J_YES` with value `0`, the function code for an unconditional jump instruction. Modify the branch prediction logic so that it predicts conditional jumps as being taken when valC $<$ valP (backward branch) and as being not-taken when valC $\geq$ valP (forward branch). Continue to predict unconditional jumps and `call` as being taken. You will need to devise a way to get both valC and valP to pipeline register M to recover from mispredicted branches. See the lab material for directions on how to generate a simulator for your solution and how to test it.

Homework Problem 41 ♦♦♦:

In our design of PIPE, we generate a stall whenever one instruction performs a *load*, reading a value from memory into a register, and the next instruction requires has this register as a source operand. When the source gets used in the execute stage, this stalling is the only way to avoid a hazard.

For cases where the second instruction stores the source operand to memory, such as with an `rmmovl` or `pushl` instruction, this stalling is not necessary. Consider the following code examples:

```
1     mrmovl 0(%ecx),%edx   # Load  1
2     pushl  %edx           # Store 1
3     nop
4     popl %edx             # Load  2
5     rmmovl %eax,0(%edx)   # Store 2
```

In lines 1 and 2, the `mrmovl` instruction reads a value from memory into `%edx`, and the `pushl` instruction then pushes this value onto the stack. Our design for PIPE would stall the `pushl` instruction to avoid a load/use hazard. Observe, however, that the value of `%edx` is not required by the `pushl` instruction until it reaches the memory stage. We can add an additional bypass path, as diagrammed in Figure 69, to forward the memory output (signal m_valM) to the valA field in pipeline register M. On the next clock cycle, this forwarded value can then be written to memory. This technique is known as *load forwarding*

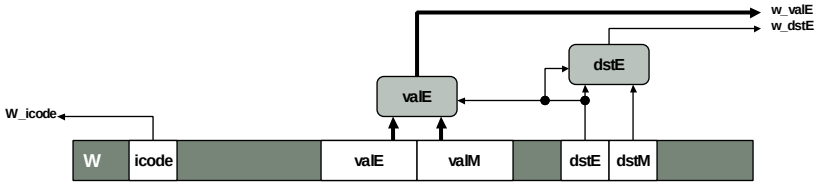
Note that the second example (lines 4 and 5) in the code sequence above cannot make use of load forwarding. The value loaded by the `popl` instruction is used as part of the address computation by the next instruction, and this value is required in the execute stage rather than the memory stage.

A. Write a formula describing the detection condition for a load/use hazard, similar to the one given in Figure 64, except that it will not cause a stall in cases where load forwarding can be used.

B. The file `pipe-lf.hcl` contains a modified version of the control logic for PIPE. It contains the definition of a signal new_M_valA to implement the block labeled "Fwd A" in Figure 69. It also has the conditions for a load/use hazard in the pipeline control logic set to `0`, and so the pipeline control logic will not detect any forms of load/use hazards. Modify this HCL description to implement load forwarding. See the lab material for directions on how to generate a simulator for your solution and how to test it.

Homework Problem 42 ♦♦♦:

Our pipelined design is a bit unrealistic in that we have two write ports for the register file, but only the `popl` instruction requires two simultaneous writes to the register file. The other instructions could therefore use a single write port, sharing this for writing valE and valM. The following figure shows a modified version of the write-back logic, in which we merge the write-back register IDs (W_dstE and W_dstM) into a single signal w_dstE, and the write-back values (W_valE and W_valM) into a single signal w_valE:



The logic for performing the merges is written in HCL as follows:

```
int w_dstE = [
        ## writing from valM
        W_dstM != RNONE : W_dstM;
        1: W_dstE;
];
int w_valE = [
        W_dstM != RNONE : W_valM;
        1: W_valE;
];
```

The control for these multiplexors is determined by dstE—when it indicates there is some register, then it selects the value for port E, and otherwise it selects the value for port M.

In the simulation model, we can then disable register port M, as shown by the following HCL code:

```
int w_dstM = RNONE;
int w_valM = 0;
```

The challenge then becomes to devise a way to handle `popl`. One method is to use the control logic to dynamically process the instruction `popl` rA so that it has the same effect as the two-instruction sequence

```
    iaddl $4, %esp
    mrmovl -4(%esp), rA
```

See homework problem 32 for a description of the `iaddl` instruction. Note the ordering of the two instructions to make sure `popl %esp` works properly. You can do this by having the logic in the decode stage treat `popl` the same as it would the `iaddl` listed above, except that it predicts the next PC to be equal to the current PC. On the next cycle, the `popl` instruction is refetched, but the instruction code is converted to a special value `IPOP2`. This is treated as a special instruction that has the same behavior as the `mrmovl` instruction listed above.

The file `pipe-1w.hcl` contains the modified write-port logic described above. It contains a declaration of the constant `IPOP2` having hexadecimal value E. It also contains the definition of a signal new_D_icode that generates the icode field for pipeline register D. This definition can be modified to insert the instruction code `IPOP2` the second time the `popl` instruction is fetched. The HCL file also contains a declaration of the signal f_pc, the value of the program counter generated in the fetch stage by the block labeled "Select PC" (Figure 56).

Modify the control logic in this file to process `popl` instructions in the manner we have described. See the lab material for directions on how to generate a simulator for your solution and how to test it.

References

[1] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1999. Order Number 243191.
Also available at `http://developer.intel.com/`.

[2] John H. Crawford. The i486 CPU: Executing instructions in one clock cycle. *IEEE Micro*, 10(1):27–36, February 1990.

[3] L. Gwennap. New algorithm improves branch prediction. *Microprocessor Report*, 9(4), March 1995.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Third Edition*. Morgan-Kaufmann, San Francisco, 2002.

[5] R. H. Katz. *Contemporary Logic Design*. Addison-Wesley, Reading, MA, 1994.

[6] B. Shriver and B. Smith. *The Anatomy of a High-Performance Microprocessor: A Systems Perspective*. IEEE Computer Society, Los Alamitos, CA, 1998.

Practice Problem solutions

Problem 1 Solution: [Pg. 9]

Encoding instructions by hand is rather tedious, but it will solidify your understanding of the idea that assembly code gets turned into byte sequences by the assembler. In the following output from our Y86 assembler, each line shows an address and a byte sequence that starts at that address.

```
1   0x100:              | .pos 0x100  # Start generating code at address 0x100
2   0x100: 30830f000000 |     irmovl $15,%ebx
3   0x106: 2031         |     rrmovl %ebx,%ecx
4   0x108:              | loop:
5   0x108: 4013fdffffff |     rmmovl %ecx,-3(%ebx)
6   0x10e: 6031         |     addl   %ebx,%ecx
7   0x110: 7008010000   |     jmp loop
```

Several features of this encoding are worth noting:

- Decimal 15 (line 2) has hex representation `0x0000000f`. Writing the bytes in reverse order gives `0f 00 00 00`.
- Decimal -3 (line 5) has hex representation `0xfffffffd`. Writing the bytes in reverse order gives `fd ff ff ff`.
- The code starts at address `0x100`. The first instruction requires 6 bytes, while the second requires 2. Thus, the loop target will be `0x00000108`. Writing these bytes in reverse order gives `08 01 00 00`.

Problem 2 Solution: [Pg. 10]

Decoding a byte sequence by hand helps you understand the task faced by a processor. It must read byte sequences and determine what instructions are to be executed. In the following, we show the assembly code used to generate each of the byte sequences. To the left of the assembly code, you can see that address and byte sequence for each instruction.

A. Some operations with immediate data and address displacements.

```
0x100: 3083fcffffff |     irmovl $-4,%ebx
0x106: 406300080000 |     rmmovl %esi,0x800(%ebx)
0x10c: 10           |     halt
```

B. Code including a function call.

```
0x200: a068         |     pushl %esi
0x202: 8008020000   |     call proc
0x207: 10           |     halt
0x208:              | proc:
0x208: 30830a000000 |     irmovl $10,%ebx
0x20e: 90           |     ret
```

C. Code containing illegal instruction specifier byte `0xf0`.

```
0x300: 505407000000 |     mrmovl 7(%esp),%ebp
0x306: 00           |     nop
0x307: f0           | .byte 0xf0  # invalid instruction code
0x308: b018         |     popl %ecx
```

D. Code containing a jump operation.

```
0x400:              | loop:
0x400: 6113         |     subl %ecx, %ebx
0x402: 7300040000   |     je loop
0x407: 10           |     halt
```

E. Code containing an invalid second byte in a `pushl` instruction:

```
0x500: 6362         |     xorl %esi,%edx
0x502: a0           |     .byte 0xa0  # pushl instruction code
0x503: 80           |     .byte 0x80  # Invalid register specifier byte
```

Problem 3 Solution: [Pg. 16]

As suggested in the problem, we adapted the code generated by GCC for an IA32 machine:

```
        # int Sum(int *Start, int Count)
rSum:   pushl %ebp
        rrmovl %esp,%ebp
        pushl %ebx           # Save value of %ebx
        mrmovl 8(%ebp),%ebx  # Get Start
        mrmovl 12(%ebp),%eax # Get Count
        andl %eax,%eax       # Test value of Count
        jle L38              # If <= 0, goto zreturn
        irmovl $-1,%edx
        addl %edx,%eax       # Count--
```

```
        pushl %eax           # Push Count
        irmovl $4,%edx
        rrmovl %ebx,%eax
        addl %edx,%eax
        pushl %eax           # Push Start+1
        call rSum            # Sum(Start+1, Count-1)
        mrmovl (%ebx),%edx
        addl %edx,%eax       # Add *Start
        jmp L39              # goto done
L38:    xorl %eax,%eax       # zreturn:
L39:    mrmovl -4(%ebp),%ebx # done: Restore %ebx
        rrmovl %ebp,%esp     # Deallocate stack frame
        popl %ebp            # Restore %ebp
        ret
```

Problem 4 Solution: [Pg. 16]

Although it is hard to imagine any practical use for this particular instruction, it is important when designing a system to avoid any ambiguities in the specification. We want to determine a reasonable convention for the instruction's behavior and make sure each of our implementations adheres to this convention.

The `subl` instruction in this test compares the starting value of `%esp` to the value pushed on a stack. The fact that the result of this subtraction is zero implies that the old value of `%esp` gets pushed.

Problem 5 Solution: [Pg. 17]

It is even more difficult to imagine why anyone would want to pop to the stack pointer. Still, we should decide on a convention and stick with it. This code sequence pushes `tval` onto the stack, pops to `%esp`, and returns the popped value. Since the result equals `tval`, we can deduce that `popl %esp` should set the stack pointer to the value read from memory. It is therefore equivalent to the instruction `mrmovl 0(%esp),%esp`.

Problem 6 Solution: [Pg. 19]

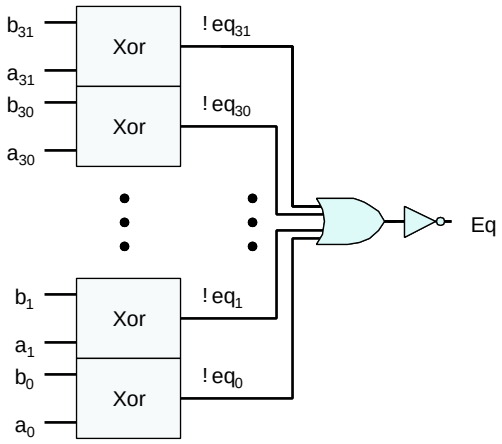
The EXCLUSIVE-OR function requires that the two bits have opposite values:

```
bool eq = (!a && b) || (a && !b);
```

In general, the signals `eq` and `xor` will be complements of each other. That is, one will equal 1 whenever the other is 0.

Problem 7 Solution: [Pg. 21]

The outputs of the EXCLUSIVE-OR circuits will be the complements of the bit equality values. Using DeMorgan's laws (Figure **??**), we can implement AND using OR and NOT, yielding the following circuit:



Problem 8 Solution: [Pg. 24]

This design is a simple variant of the one to find the minimum of the three inputs:

```
int Med3 = [
        A <= B && B <= C : B;
        B <= A && A <= C : A;
        1                : C;
];
```

Problem 9 Solution: [Pg. 31]

These exercises help make the stage computations more concrete. We can see from the object code that this instruction is located at address `0x00e`. It consists of six bytes, with the first two being `0x30` and `0x84`. The last four bytes are a byte-reversed version of `0x00000080` (decimal `128`).

Stage	Generic	Specific
	`irmovl V, rB`	`irmovl $128, %esp`
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valC $\leftarrow$ M_4[PC + 2] valP $\leftarrow$ PC + 6	icode:ifun $\leftarrow$ M_1[0x00e] = 3:0 rA:rB $\leftarrow$ M_1[0x00f] = 8:4 valC $\leftarrow$ M_4[0x010] = 128 valP $\leftarrow$ 0x00e + 6 = 0x014
Decode		
Execute	valE $\leftarrow$ 0 + valC	valE $\leftarrow$ 0 + 128 = 128
Memory		
Write back	R[rB] $\leftarrow$ valE	R[%esp] $\leftarrow$ valE = 128
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP = 0x014

This instruction sets register %esp to 128 and increments the PC by 6.

Problem 10 Solution: [Pg. 33]

We can see that the instruction is located at address 0x01c and consists of two bytes with values 0xb0 and 0x08. Register %esp was set to 124 by the pushl instruction (line 6), which also stored 9 at this memory location.

Stage	Generic	Specific
	popl rA	popl %eax
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC+1] valP $\leftarrow$ PC + 2	icode:ifun $\leftarrow$ M_1[0x01c] = b:0 rA:rB $\leftarrow$ M_1[0x01d] = 0:8 valP $\leftarrow$ 0x01c + 2 = 0x01e
Decode	valA $\leftarrow$ R[%esp] valB $\leftarrow$ R[%esp]	valA $\leftarrow$ R[%esp] = 124 valB $\leftarrow$ R[%esp] = 124
Execute	valE $\leftarrow$ valB + 4	valE $\leftarrow$ 124 + 4 = 128
Memory	valM $\leftarrow$ M_4[valA]	valM $\leftarrow$ M_4[124] = 9
Write back	R[%esp] $\leftarrow$ valE R[rA] $\leftarrow$ valM	R[%esp] $\leftarrow$ 128 R[%eax] $\leftarrow$ 9
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ 0x01e

The instruction sets %eax to 9, sets %esp to 128, and increments the PC by 2.

Problem 11 Solution: [Pg. 34]

Tracing the steps listed in Figure 18 with rA equal to %esp, we can see that in the memory stage, the instruction will store valA, the original value of the stack pointer, to memory, just as we found for IA32.

Problem 12 Solution: [Pg. 34]

Tracing the steps listed in Figure 18 with rA equal to %esp, we can see that both of the write-back operations will update %esp. Since the one writing valM would occur last, the net effect of the instruction will be to write the value read from memory to %esp, just as we saw for IA32.

Problem 13 Solution: [Pg. 35]

We can see that this instruction is located at address 0x023 and is 5 bytes long. The first byte has value 0x80, while the last four are a byte-reversed version of 0x00000029, the call target. The stack pointer was set to 128 by the popl instruction (line 7).

Stage	Generic	Specific
	`call Dest`	`call 0x029`
Fetch	icode:ifun $\leftarrow$ M_1[PC] valC $\leftarrow$ M_4[PC + 1] valP $\leftarrow$ PC + 5	icode:ifun $\leftarrow$ M_1[0x023] = 8:0 valC $\leftarrow$ M_4[0x024] = 0x029 valP $\leftarrow$ 0x023 + 5 = 0x028
Decode	 valB $\leftarrow$ R[%esp]	 valB $\leftarrow$ R[%esp] = 128
Execute	valE $\leftarrow$ valB + −4	valE $\leftarrow$ 128 + −4 = 124
Memory	M_4[valE] $\leftarrow$ valP	M_4[124] $\leftarrow$ 0x028
Write back	R[%esp] $\leftarrow$ valE	R[%esp] $\leftarrow$ 124
PC update	PC $\leftarrow$ valC	PC $\leftarrow$ 0x029

The effect of this instruction is to set `%esp` to 124, to store `0x028` (the return address) at this memory address, and to set the PC to `0x029` (the call target).

Problem 14 Solution: [Pg. 45]

All of the HCL code in this and other practice problems is straightforward, but trying to generate it yourself will help you think about the different instructions and how they are processed. For this problem, we can simply look at the set of Y86 instructions (Figure 2) and determine which have a constant field.

```
bool need_valC =
        icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
```

Problem 15 Solution: [Pg. 46]

This code is similar to the code for srcA.

```
int srcB = [
        icode in { IOPL, IRMMOVL, IMRMOVL  } : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't need register
];
```

Problem 16 Solution: [Pg. 47]

This code is similar to the code for dstE.

```
int dstM = [
        icode in { IMRMOVL, IPOPL } : rA;
        1 : RNONE;  # Don't need register
];
```

Problem 17 Solution: [Pg. 47]

As we found in practice problem 12, we want the write via the M port to take priority over the the write via the E port in order to store the value read from memory into `%esp`.

Problem 18 Solution: [Pg. 48]

This code is similar to the code for aluA.

```
int aluB = [
        icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
                        IPUSHL, IRET, IPOPL } : valB;
        icode in { IRRMOVL, IIRMOVL } : 0;
        # Other instructions don't need ALU
];
```

Problem 19 Solution: [Pg. 49]

This code is similar to the code for mem_addr.

```
int mem_data = [
        # Value from register
        icode in { IRMMOVL, IPUSHL } : valA;
        # Return PC
        icode == ICALL : valP;
        # Default: Don't write anything
];
```

Problem 20 Solution: [Pg. 49]

This code is similar to the code for mem_read.

```
bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };
```

Problem 21 Solution: [Pg. 58]

This problem is an interesting exercise in trying to find the optimal balance among a set of partitions. It provides a number of opportunities to compute throughputs and latencies in pipelines.

A. For a two-stage pipeline, the best partition would be to have blocks A, B, and C in the first stage and D, E, and F in the second. The first stage has a delay 170 ps, giving a total cycle time of $170 + 20 = 190$ picoseconds. We therefore have a throughput of 5.26 GOPS and a latency of 380 ps.

B. For a three-stage pipeline, we should have blocks A and B in the first stage, blocks C and D in the second, and blocks E and F in the third. The first two stages have a delay of 110 ps, giving a total cycle time of 130 ps and a throughput of 7.69 GOPS. The latency is 390 ps.

C. For a four-stage pipeline, we should have block A in the first stage, blocks B and C in the second, block D in the third, and blocks E and F in the fourth. The second stage requires 90 ps, giving a total cycle time of 110 ps and a throughput of 9.09 GOPS. The latency is 440 ps.

D. The optimal design would be a five-stage pipeline, with each block in its own stage, except that the fifth stage has blocks E and F. The cycle time is $80 + 20 = 100$ picoseconds, for a throughput of around 10.00 GOPS and a latency of 500 ps. Adding more stages would not help, since we cannot run the pipeline any faster than one cycle every 100 ps.

Problem 22 Solution: [Pg. 60]

In the limit, we could have a pipeline where each computational block has a delay of ϵ ns. The clock period would be $\epsilon + 20$ picoseconds, giving throughput $1000/(\epsilon + 20)$. As the number of stages grows arbitrarily large, ϵ would tend toward 0, giving a throughput of 50.00 GOPS.

Problem 23 Solution: [Pg. 86]

This code simply involves prefixing the signal names in the code for SEQ with "`D_`"

```
int new_E_dstE = [
        D_icode in { IRRMOVL, IIRMOVL, IOPL} : D_rB;
        D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't need register
];
```

Problem 24 Solution: [Pg. 90]

The `rrmovl` instruction (line 5) would stall for one cycle due to a load-use hazard caused by the `popl` instruction (line 4). As it enters the decode stage, the `popl` instruction would be in the memory stage, giving both M_dstE and M_dstM equal to `%esp`. If the two cases were reversed, then the write back from M_valE would take priority, causing the incremented stack pointer to be passed as the argument to the `rrmovl` instruction. This would not be consistent with the convention for handling `popl %esp` determined in practice problem 5.

Problem 25 Solution: [Pg. 90]

This problem lets you experience one of the important tasks in processor design—devising test programs for a new processor. In general, we should have test programs that will exercise all of the different hazard possibilities and will generate incorrect results if some dependency is not handled properly.

For this example, we can use a slightly modified version of the program shown in practice problem 24:

```
1     irmovl $5, %edx
2     irmovl $0x100,%esp
3     rmmovl %edx,0(%esp)
4     popl %esp
5     nop
6     nop
7     rrmovl %esp,%eax
```

The two `nop` instructions will cause the `popl` instruction to be in the write-back stage when the `rrmovl` instruction is in the decode stage. If the two forwarding sources in the write-back stage are given the wrong priority, then register `%eax` will be set to the incremented program counter rather than the value read from memory.

Problem 26 Solution: [Pg. 90]

This logic only needs to check the five forwarding sources:

```
int new_E_valB = [
        d_srcB == E_dstE : e_valE;    # Forward valE from execute
        d_srcB == M_dstM : m_valM;    # Forward valM from memory
        d_srcB == M_dstE : M_valE;    # Forward valE from memory
        d_srcB == W_dstM : W_valM;    # Forward valM from write back
        d_srcB == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalB;  # Use value read from register file
];
```

Problem 27 Solution: [Pg. 97]

The following test program is designed to set up control combination A (Figure 67) and detect whether something goes wrong:

```
 1 # Code to generate a combination of not-taken branch and ret
 2         irmovl Stack, %esp
 3         irmovl rtnp,%eax
 4         pushl %eax          # Set up return pointer
 5         xorl %eax,%eax      # Set Z condition code
 6         jne target          # Not taken (First part of combination)
 7         irmovl $1,%eax      # Should execute this
 8         halt
 9 target: ret                 # Second part of combination
10         irmovl $2,%ebx      # Should not execute this
11         halt
12 rtnp:   irmovl $3,%edx      # Should not execute this
13         halt
14 .pos 0x40
15 Stack:
```

This program is designed so that if something goes wrong, for example the `ret` instruction is actually executed, then the program will execute one of the extra `irmovl` instructions and then halt. Thus, an error in the pipeline would cause some register to be updated incorrectly. This code illustrates the care required to implement a test program. It must set up a potential error condition and then detect whether or not an error occurs.

Problem 28 Solution: [Pg. 98]

The following test program is designed to set up control combination B (Figure 67). The simulator will detect a case where the bubble and stall control signals for a pipeline register are both set to 0, and so our

test program need only set up the combination for it to be detected. The biggest challenge is to make the program do something sensible when handled correctly.

```
 1 # Test instruction that modifies %esp followed by ret
 2         irmovl mem,%ebx
 3         mrmovl  0(%ebx),%esp # Sets %esp to point to return point
 4         ret                  # Returns to return point
 5         halt                 #
 6 rtnpt:  irmovl $5,%esi       # Return point
 7         halt
 8 .pos 0x40
 9 mem:    .long stack          # Holds desired stack pointer
10 .pos 0x50
11 stack:  .long rtnpt          # Top of stack: Holds return point
```

This program uses two initialized word in memory. The first word (`mem`) holds the address of the second (`stack`—the desired stack pointer). The second word holds the address of the desired return point for the `ret` instruction. The program loads the stack pointer into `%esp` and executes the `ret` instruction.

Problem 29 Solution: [Pg. 99]

From Figure 66, we can see that pipeline register D must be stalled for a load/use hazard.

```
bool D_stall =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVL, IPOPL } &&
         E_dstM in { d_srcA, d_srcB };
```

Problem 30 Solution: [Pg. 100]

From Figure 66, we can see that pipeline register E must be set to bubble for a load/use hazard or for a mispredicted branch:

```
bool E_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Bch) ||
        # Conditions for a load/use hazard
        E_icode in { IMRMOVL, IPOPL } &&
           E_dstM in { d_srcA, d_srcB};
```

Problem 31 Solution: [Pg. 101]

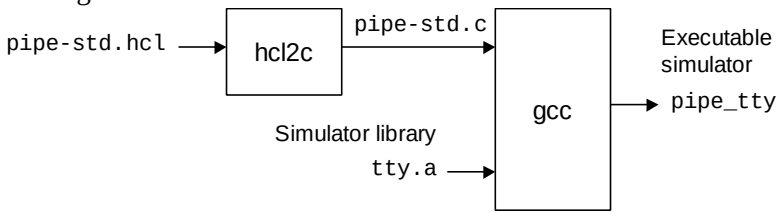
We would then have a misprediction frequency of 0.35, giving $mp = 0.20 \times 0.35 \times 2 = 0.14$, giving an overall CPI of 1.25. This seems like a fairly marginal gain, but it would be worthwhile if the cost of implementing the new branch prediction strategy were not too high.

HCL Descriptions of the Processors

A HCL Reference Manual

In Chapter **??**, we use HCL (short for "Hardware Control Language") to describe the control logic portions of several processor designs. HCL has some of the features of a hardware description language, allowing users to describe Boolean functions and word-level selection operations. On the other hand, it lacks many features found in true HDLs, such as ways to declare registers and other storage elements; looping and conditional constructs; module definition and instantiation capabilities; and bit extraction and insertion operations.

HCL is really just a language for generating a very stylized form of C code. All of the block definitions in an HCL file get converted to C functions by a program HCL2C (for "HCL to C"). These functions are then compiled and linked with library code implementing the other simulator functions to generate an executable simulation program, as diagrammed below:



This diagram shows the files used to generate the text version of the pipeline simulator.

It would be possible to describe the behavior of the control logic directly in C, rather than writing HCL and translating this to C. The advantage of the HCL route is that we more clearly separate the functionality of the hardware from the inner workings of the simulator.

HCL supports just two data types: `bool` (for "Boolean") signals are either 0 or 1, while `int` (for "integer") signals are equivalent to `int` values in C. Data type `int` is used for all types of multi-bit signals, such as words, register IDs, and instruction codes. When converted to C, both data types are represented as `int` data, but a value of type `bool` will only equal 0 or 1.

A.1 Signal Declarations

Expressions in HCL can reference named *signals* of type integer or Boolean. The signal names must start with a letter (`a`–`z` or `A`–`Z`), followed by any number of letters, digits, or underscores (`_`). Signal names are case sensitive. The Boolean and integer signal names used in HCL Boolean and integer expressions are really just aliases for C expressions. The declaration of a signal also defines the associated C expression. A signal declaration has one of the following forms:

```
boolsig  name  'C-expr'
intsig   name  'C-expr'
```

where *C-expr* can be an arbitrary C expression, except that it cannot contain a single quote (`'`) or a newline character (`\n`). When generating C code, HCL2C will replace any signal name with the corresponding C

Syntax	Meaning
`0`	Logic value 0
`1`	Logic value 1
name	Named Boolean signal
int-expr `in` {*int-expr*$_1$, *int-expr*$_2$, ..., *int-expr*$_k$}	Set membership test
int-expr$_1$ `==` *int-expr*$_2$	Equality test
int-expr$_1$ `!=` *int-expr*$_2$	Not equal test
int-expr$_1$ `<` *int-expr*$_2$	Less than test
int-expr$_1$ `<=` *int-expr*$_2$	Less than or equal test
int-expr$_1$ `>` *int-expr*$_2$	Greater than test
int-expr$_1$ `>=` *int-expr*$_2$	Greater than or equal test
`!` *bool-expr*	NOT
bool-expr$_1$ `&&` *bool-expr*$_2$	AND
bool-expr$_1$ `||` *bool-expr*$_2$	OR

Figure 70: **HCL Boolean expressions.** These expressions evaluate to 0 or 1. The operations are listed in descending order of precedence, where those within each group have equal precedence.

expression.

A.2 Quoted Text

Quoted text provides a mechanism to pass text directly through HCL2C into the generated C file. This can be used to insert variable declarations, `include` statements, and other things generally found in C files. The general form is:

`quote` `'`*string*`'`

where *string* can be any string that does not contain single quotes (`'`) or newline characters (`\n`).

A.3 Expressions and Blocks

There are two types of expressions: Boolean and integer, which we refer to in our syntax descriptions as *bool-expr* and *int-expr*, respectively. Figure 70 lists the different types of Boolean expressions. They are listed in descending order of precedence, with the operations within each group (groups are separated by horizontal lines) having equal precedence. Parentheses can be used to override the normal operator precedence.

At the top level are the constant values `0` and `1` and named Boolean signals. Next in precedence are expressions that have integer arguments but yield Boolean results. The set membership test compares the value of the first integer expression *int-expr* to the values of each of the integer expressions comprising the set {*int-expr*$_1$, ... *int-expr*$_k$}, yielding 1 if any matching value is found. The relational operators compare two integer expressions, generating 1 when the relation holds and 0 when it does not.

The remaining expressions in Figure 70 consist of formulas using Boolean connectives (`!` for NOT, `&&` for AND. and `||` for OR).

There are just three types of integer expressions: numbers, named integer signals, and case expressions. Numbers are written in decimal notation and can be negative. Named integer signals use the naming rules described earlier. Case expressions have the following general form:

```
[
    bool-expr_1  :  int-expr_1
    bool-expr_2  :  int-expr_2
                 ⋮
    bool-expr_k  :  int-expr_k
]
```

The expression contains a series of cases, where each case i consists of a Boolean expression $bool\text{-}expr_i$, indicating whether this case should be selected, and an integer expression $int\text{-}expr_i$, indicating the value resulting for this case. In evaluating a case expression, the Boolean expressions are conceptually evaluated in sequence. When one of them yields 1, the value of the corresponding integer expression is returned as the case expression value. If no Boolean expression evaluates to 1, then the value of the case expression is 0. One good programming practice is to have the last Boolean expression be `1`, guaranteeing at least one matching case.

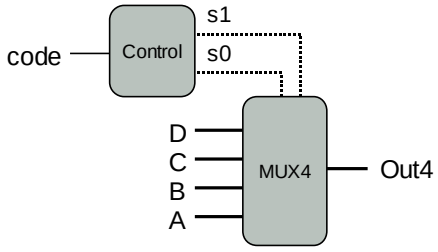
HCL expressions are used to define the behavior of a block of combinational logic. A block definition has one of the following forms:

```
bool name  =  bool-expr;
 int name  =  int-expr;
```

where the first form defines a Boolean block, while the second defines a word-level block. For a block declared with $name$ as its name, HCL2C generates a function `gen_`$name$. This function has no arguments, and it returns a result of type `int`.

A.4 HCL Example

The following example shows a complete HCL file. The C code generated by processing it with HCL2C is completely self-contained. It can be compiled and run using command line arguments for the input signals. More typically, HCL files define just the control part of a simulation model. The generated C code is then compiled and linked with other code to form the executable simulator. We show this example just to give a concrete example of HCL. The circuit is based on the `MUX4` circuit described in Section 2.4, with the following structure:




```
## Simple example of an HCL file.
## This file can be converted to C using hcl2c, and then compiled.

## In this example, we will generate the MUX4 circuit shown in
## Section 2.4.  It consists of a control block that generates
## bit-level signals s1 and s0 from the input signal code,
## and then uses these signals to control a 4-way multiplexor
## with data inputs A, B, C, and D.

## This code is embedded in a C program that reads
## the values of code, A, B, C, and D from the command line
## and then prints the circuit output

## Information that is inserted verbatim into the C file
quote '#include <stdio.h>'
quote '#include <stdlib.h>'
quote 'int code_val, s0_val, s1_val;'
quote 'char **data_names;'

## Declarations of signals used in the HCL description and
## the corresponding C expressions.
boolsig s0 's0_val'
boolsig s1 's1_val'
intsig code 'code_val'
intsig  A 'atoi(data_names[0])'
intsig  B 'atoi(data_names[1])'
intsig  C 'atoi(data_names[2])'
intsig  D 'atoi(data_names[3])'

## HCL descriptions of the logic blocks
bool s1 = code in { 2, 3 };

bool s0 = code in { 1, 3 };

int Out4 = [
        !s1 && !s0 : A; # 00
        !s1        : B; # 01
        s1 && !s0  : C; # 10
        1          : D; # 11
];

```

```
42 ## More information inserted verbatim into the C code to
43 ## compute the values and print the output
44 quote 'int main(int argc, char *argv[]) {'
45 quote '  data_names = argv+2;'
46 quote '  code_val = atoi(argv[1]);'
47 quote '  s1_val = gen_s1();'
48 quote '  s0_val = gen_s0();'
49 quote '  printf("Out = %d\n", gen_Out4());'
50 quote '  return 0;'
51 quote '}'
```

This file defines Boolean signals `s0` and `s1` and integer signal `code` to be aliases for references to global variables `s0_val`, `s1_val`, and `code_val`. It declares integer signals `A`, `B`, `C`, and `D`, where the corresponding C expressions apply the standard library function `atoi` to strings passed as command line arguments.

The definition of the block named `s1` generates the following C code:

```
int gen_s1()
{
    return ((code_val) == 2||(code_val) == 3);
}
```

As can be seen here, set membership testing is implemented as a series of comparisons, and that every reference to signal `code` is replaced by the C expression `code_val`.

Note that there is no direct relation between the signal `s1` declared on line 23 of the HCL file, and the block named `s1` declared on line 31. One is an alias for a C expression, while the other generates a function named `gen_s1`.

The quoted text at the end generates the following main function:

```
int main(int argc, char *argv[]) {
  data_names = argv+2;
  code_val = atoi(argv[1]);
  s1_val = gen_s1();
  s0_val = gen_s0();
  printf("Out = %d\n", gen_Out4());
  return 0;
}
```

The main function calls the functions `gen_s1`, `gen_s0`, and `gen_Out4` that were generated from the block definitions. We can also see how the C code must define the sequencing of block evaluations and the setting of the values used in the C expressions representing the different signal values.

B SEQ

code/arch/seq/seq-std.hcl

```
1  ##################################################################
2  #  HCL Description of Control for Single Cycle Y86 Processor SEQ   #
3  #  Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2002       #
4  ##################################################################
5
6  ##################################################################
7  #    C Include's.  Don't alter these                             #
8  ##################################################################
9
10 quote '#include <stdio.h>'
11 quote '#include "isa.h"'
12 quote '#include "sim.h"'
13 quote 'int sim_main(int argc, char *argv[]);'
14 quote 'int gen_pc(){return 0;}'
15 quote 'int main(int argc, char *argv[])'
16 quote '  {plusmode=0;return sim_main(argc,argv);}'
17
18 ##################################################################
19 #    Declarations.  Do not change/remove/delete any of these       #
20 ##################################################################
21
22 ##### Symbolic representation of Y86 Instruction Codes #############
23 intsig INOP     'I_NOP'
24 intsig IHALT    'I_HALT'
25 intsig IRRMOVL  'I_RRMOVL'
26 intsig IIRMOVL  'I_IRMOVL'
27 intsig IRMMOVL  'I_RMMOVL'
28 intsig IMRMOVL  'I_MRMOVL'
29 intsig IOPL     'I_ALU'
30 intsig IJXX     'I_JMP'
31 intsig ICALL    'I_CALL'
32 intsig IRET     'I_RET'
33 intsig IPUSHL   'I_PUSHL'
34 intsig IPOPL    'I_POPL'
35
36 ##### Symbolic representation of Y86 Registers referenced explicitly #####
37 intsig RESP     'REG_ESP'       # Stack Pointer
38 intsig RNONE    'REG_NONE'      # Special value indicating "no register"
39
40 ##### ALU Functions referenced explicitly                          #####
41 intsig ALUADD   'A_ADD'         # ALU should add its arguments
42
43 ##### Signals that can be referenced by control logic ####################
44
45 ##### Fetch stage inputs               #####
46 intsig pc 'pc'                         # Program counter
47 ##### Fetch stage computations         #####
48 intsig icode    'icode'                # Instruction control code
49 intsig ifun     'ifun'                 # Instruction function
50 intsig rA       'ra'                   # rA field from instruction
```

```
51 intsig rB       'rb'                    # rB field from instruction
52 intsig valC     'valc'                  # Constant from instruction
53 intsig valP     'valp'                  # Address of following instruction
54
55 ##### Decode stage computations         #####
56 intsig valA     'vala'                  # Value from register A port
57 intsig valB     'valb'                  # Value from register B port
58
59 ##### Execute stage computations        #####
60 intsig valE     'vale'                  # Value computed by ALU
61 boolsig Bch     'bcond'                 # Branch test
62
63 ##### Memory stage computations         #####
64 intsig valM     'valm'                  # Value read from memory
65
66
67 ################################################################################
68 #    Control Signal Definitions.                                               #
69 ################################################################################
70
71 ################ Fetch Stage     ###################################
72
73 # Does fetched instruction require a regid byte?
74 bool need_regids =
75         icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
76                        IIRMOVL, IRMMOVL, IMRMOVL };
77
78 # Does fetched instruction require a constant word?
79 bool need_valC =
80         icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
81
82 bool instr_valid = icode in
83         { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
84                IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
85
86 ################ Decode Stage    ###################################
87
88 ## What register should be used as the A source?
89 int srcA = [
90         icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : rA;
91         icode in { IPOPL, IRET } : RESP;
92         1 : RNONE; # Don't need register
93 ];
94
95 ## What register should be used as the B source?
96 int srcB = [
97         icode in { IOPL, IRMMOVL, IMRMOVL  } : rB;
98         icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
99         1 : RNONE;  # Don't need register
100 ];
```

```
101
102 ## What register should be used as the E destination?
103 int dstE = [
104         icode in { IRRMOVL, IIRMOVL, IOPL} : rB;
105         icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
106         1 : RNONE;  # Don't need register
107 ];
108
109 ## What register should be used as the M destination?
110 int dstM = [
111         icode in { IMRMOVL, IPOPL } : rA;
112         1 : RNONE;  # Don't need register
113 ];
114
115 ################ Execute Stage   ###################################
116
117 ## Select input A to ALU
118 int aluA = [
119         icode in { IRRMOVL, IOPL } : valA;
120         icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
121         icode in { ICALL, IPUSHL } : -4;
122         icode in { IRET, IPOPL } : 4;
123         # Other instructions don't need ALU
124 ];
125
126 ## Select input B to ALU
127 int aluB = [
128         icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
129                        IPUSHL, IRET, IPOPL } : valB;
130         icode in { IRRMOVL, IIRMOVL } : 0;
131         # Other instructions don't need ALU
132 ];
133
134 ## Set the ALU function
135 int alufun = [
136         icode == IOPL : ifun;
137         1 : ALUADD;
138 ];
139
140 ## Should the condition codes be updated?
141 bool set_cc = icode in { IOPL };
142
143 ################ Memory Stage    ###################################
144
145 ## Set read control signal
146 bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
147
148 ## Set write control signal
149 bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };
150
```

```
151 ## Select memory address
152 int mem_addr = [
153         icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
154         icode in { IPOPL, IRET } : valA;
155         # Other instructions don't need address
156 ];
157
158 ## Select memory input data
159 int mem_data = [
160         # Value from register
161         icode in { IRMMOVL, IPUSHL } : valA;
162         # Return PC
163         icode == ICALL : valP;
164         # Default: Don't write anything
165 ];
166
167 ################ Program Counter Update ############################
168
169 ## What address should instruction be fetched at
170
171 int new_pc = [
172         # Call.  Use instruction constant
173         icode == ICALL : valC;
174         # Taken branch.  Use instruction constant
175         icode == IJXX && Bch : valC;
176         # Completion of RET instruction.  Use value from stack
177         icode == IRET : valM;
178         # Default: Use incremented PC
179         1 : valP;
180 ];
```

code/arch/seq/seq-std.hcl

C SEQ+

code/arch/seq/seq+-std.hcl

```
  1 ##################################################################
  2 #  HCL Description of Control for Single Cycle Y86 Processor SEQ+  #
  3 #  Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2002       #
  4 ##################################################################
  5
  6 ##################################################################
  7 #    C Include's.  Don't alter these                             #
  8 ##################################################################
  9
 10 quote '#include <stdio.h>'
 11 quote '#include "isa.h"'
```

```
12 quote '#include "sim.h"'
13 quote 'int sim_main(int argc, char *argv[]);'
14 quote 'int gen_new_pc(){return 0;}'
15 quote 'int main(int argc, char *argv[])'
16 quote '  {plusmode=1;return sim_main(argc,argv);}'
17
18 ##################################################################
19 #    Declarations.  Do not change/remove/delete any of these       #
20 ##################################################################
21
22 ##### Symbolic representation of Y86 Instruction Codes #############
23 intsig INOP     'I_NOP'
24 intsig IHALT    'I_HALT'
25 intsig IRRMOVL  'I_RRMOVL'
26 intsig IIRMOVL  'I_IRMOVL'
27 intsig IRMMOVL  'I_RMMOVL'
28 intsig IMRMOVL  'I_MRMOVL'
29 intsig IOPL     'I_ALU'
30 intsig IJXX     'I_JMP'
31 intsig ICALL    'I_CALL'
32 intsig IRET     'I_RET'
33 intsig IPUSHL   'I_PUSHL'
34 intsig IPOPL    'I_POPL'
35
36 ##### Symbolic representation of Y86 Registers referenced explicitly #####
37 intsig RESP     'REG_ESP'       # Stack Pointer
38 intsig RNONE    'REG_NONE'      # Special value indicating "no register"
39
40 ##### ALU Functions referenced explicitly                            #####
41 intsig ALUADD   'A_ADD'         # ALU should add its arguments
42
43 ##### Signals that can be referenced by control logic ####################
44
45 ##### PC stage inputs                  #####
46
47 ## All of these values are based on those from previous instruction
48 intsig  pIcode 'prev_icode'             # Instr. control code
49 intsig  pValC  'prev_valc'              # Constant from instruction
50 intsig  pValM  'prev_valm'              # Value read from memory
51 intsig  pValP  'prev_valp'              # Incremented program counter
52 boolsig pBch 'prev_bcond'              # Branch taken flag
53
54 ##### Fetch stage computations         #####
55 intsig icode    'icode'                # Instruction control code
56 intsig ifun     'ifun'                 # Instruction function
57 intsig rA       'ra'                   # rA field from instruction
58 intsig rB       'rb'                   # rB field from instruction
59 intsig valC     'valc'                 # Constant from instruction
60 intsig valP     'valp'                 # Address of following instruction
61
```

```
62 ##### Decode stage computations         #####
63 intsig valA     'vala'                  # Value from register A port
64 intsig valB     'valb'                  # Value from register B port
65
66 ##### Execute stage computations        #####
67 intsig valE     'vale'                  # Value computed by ALU
68 boolsig Bch     'bcond'                 # Branch test
69
70 ##### Memory stage computations         #####
71 intsig valM     'valm'                  # Value read from memory
72
73
74 ####################################################################
75 #    Control Signal Definitions.                                   #
76 ####################################################################
77
78 ################ Program Counter Computation ########################
79
80 # Compute fetch location for this instruction based on results from
81 # previous instruction.
82
83 int pc = [
84         # Call.  Use instruction constant
85         pIcode == ICALL : pValC;
86         # Taken branch.  Use instruction constant
87         pIcode == IJXX && pBch : pValC;
88         # Completion of RET instruction.  Use value from stack
89         pIcode == IRET : pValM;
90         # Default: Use incremented PC
91         1 : pValP;
92 ];
93
94 ################ Fetch Stage     ###################################
95
96 # Does fetched instruction require a regid byte?
97 bool need_regids =
98         icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
99                        IIRMOVL, IRMMOVL, IMRMOVL };
100
101 # Does fetched instruction require a constant word?
102 bool need_valC =
103         icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
104
105 bool instr_valid = icode in
106         { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
107                IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
108
109 ################ Decode Stage    ###################################
110
111 ## What register should be used as the A source?
```

```
112 int srcA = [
113         icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : rA;
114         icode in { IPOPL, IRET } : RESP;
115         1 : RNONE; # Don't need register
116 ];
117
118 ## What register should be used as the B source?
119 int srcB = [
120         icode in { IOPL, IRMMOVL, IMRMOVL  } : rB;
121         icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
122         1 : RNONE;  # Don't need register
123 ];
124
125 ## What register should be used as the E destination?
126 int dstE = [
127         icode in { IRRMOVL, IIRMOVL, IOPL} : rB;
128         icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
129         1 : RNONE;  # Don't need register
130 ];
131
132 ## What register should be used as the M destination?
133 int dstM = [
134         icode in { IMRMOVL, IPOPL } : rA;
135         1 : RNONE;  # Don't need register
136 ];
137
138 ################ Execute Stage   ######################################
139
140 ## Select input A to ALU
141 int aluA = [
142         icode in { IRRMOVL, IOPL } : valA;
143         icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
144         icode in { ICALL, IPUSHL } : -4;
145         icode in { IRET, IPOPL } : 4;
146         # Other instructions don't need ALU
147 ];
148
149 ## Select input B to ALU
150 int aluB = [
151         icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
152                         IPUSHL, IRET, IPOPL } : valB;
153         icode in { IRRMOVL, IIRMOVL } : 0;
154         # Other instructions don't need ALU
155 ];
156
157 ## Set the ALU function
158 int alufun = [
159         icode == IOPL : ifun;
160         1 : ALUADD;
161 ];
```

```
162
163 ## Should the condition codes be updated?
164 bool set_cc = icode in { IOPL };
165
166 ################ Memory Stage    ###################################
167
168 ## Set read control signal
169 bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
170
171 ## Set write control signal
172 bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };
173
174 ## Select memory address
175 int mem_addr = [
176         icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
177         icode in { IPOPL, IRET } : valA;
178         # Other instructions don't need address
179 ];
180
181 ## Select memory input data
182 int mem_data = [
183         # Value from register
184         icode in { IRMMOVL, IPUSHL } : valA;
185         # Return PC
186         icode == ICALL : valP;
187         # Default: Don't write anything
188 ];
```

code/arch/seq/seq+-std.hcl

D PIPE

code/arch/pipe/pipe-std.hcl

```
 1 ##################################################################
 2 #    HCL Description of Control for Pipelined Y86 Processor      #
 3 #    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2002   #
 4 ##################################################################
 5
 6 ##################################################################
 7 #    C Include's.  Don't alter these                             #
 8 ##################################################################
 9
10 quote '#include <stdio.h>'
11 quote '#include "isa.h"'
12 quote '#include "pipeline.h"'
13 quote '#include "stages.h"'
14 quote '#include "sim.h"'
```

```
15 quote 'int sim_main(int argc, char *argv[]);'
16 quote 'int main(int argc, char *argv[]){return sim_main(argc,argv);}'
17
18 ##################################################################
19 #    Declarations.  Do not change/remove/delete any of these       #
20 ##################################################################
21
22 ##### Symbolic representation of Y86 Instruction Codes #############
23 intsig INOP     'I_NOP'
24 intsig IHALT    'I_HALT'
25 intsig IRRMOVL  'I_RRMOVL'
26 intsig IIRMOVL  'I_IRMOVL'
27 intsig IRMMOVL  'I_RMMOVL'
28 intsig IMRMOVL  'I_MRMOVL'
29 intsig IOPL     'I_ALU'
30 intsig IJXX     'I_JMP'
31 intsig ICALL    'I_CALL'
32 intsig IRET     'I_RET'
33 intsig IPUSHL   'I_PUSHL'
34 intsig IPOPL    'I_POPL'
35
36 ##### Symbolic representation of Y86 Registers referenced explicitly #####
37 intsig RESP     'REG_ESP'       # Stack Pointer
38 intsig RNONE    'REG_NONE'      # Special value indicating "no register"
39
40 ##### ALU Functions referenced explicitly ##########################
41 intsig ALUADD   'A_ADD'         # ALU should add its arguments
42
43 ##### Signals that can be referenced by control logic ##############
44
45 ##### Pipeline Register F ##########################################
46
47 intsig F_predPC 'pc_curr->pc'           # Predicted value of PC
48
49 ##### Intermediate Values in Fetch Stage ###########################
50
51 intsig f_icode  'if_id_next->icode'  # Fetched instruction code
52 intsig f_ifun   'if_id_next->ifun'   # Fetched instruction function
53 intsig f_valC   'if_id_next->valc'   # Constant data of fetched instruction
54 intsig f_valP   'if_id_next->valp'   # Address of following instruction
55
56 ##### Pipeline Register D ##########################################
57 intsig D_icode 'if_id_curr->icode'      # Instruction code
58 intsig D_rA 'if_id_curr->ra'    # rA field from instruction
59 intsig D_rB 'if_id_curr->rb'    # rB field from instruction
60 intsig D_valP 'if_id_curr->valp'        # Incremented PC
61
62 ##### Intermediate Values in Decode Stage  #########################
63
64 intsig d_srcA    'id_ex_next->srca'     # srcA from decoded instruction
```

```
65 intsig d_srcB    'id_ex_next->srcb'     # srcB from decoded instruction
66 intsig d_rvalA 'd_regvala'              # valA read from register file
67 intsig d_rvalB 'd_regvalb'              # valB read from register file
68
69 ##### Pipeline Register E ########################################
70 intsig E_icode 'id_ex_curr->icode'      # Instruction code
71 intsig E_ifun  'id_ex_curr->ifun'       # Instruction function
72 intsig E_valC  'id_ex_curr->valc'       # Constant data
73 intsig E_srcA  'id_ex_curr->srca'       # Source A register ID
74 intsig E_valA  'id_ex_curr->vala'       # Source A value
75 intsig E_srcB  'id_ex_curr->srcb'       # Source B register ID
76 intsig E_valB  'id_ex_curr->valb'       # Source B value
77 intsig E_dstE 'id_ex_curr->deste'       # Destination E register ID
78 intsig E_dstM 'id_ex_curr->destm'       # Destination M register ID
79
80 ##### Intermediate Values in Execute Stage ########################
81 intsig e_valE 'ex_mem_next->vale'       # valE generated by ALU
82 boolsig e_Bch 'ex_mem_next->takebranch' # Am I about to branch?
83
84 ##### Pipeline Register M                  #####
85 intsig M_icode 'ex_mem_curr->icode'     # Instruction code
86 intsig M_ifun  'ex_mem_curr->ifun'      # Instruction function
87 intsig M_valA  'ex_mem_curr->vala'      # Source A value
88 intsig M_dstE 'ex_mem_curr->deste'      # Destination E register ID
89 intsig M_valE  'ex_mem_curr->vale'      # ALU E value
90 intsig M_dstM 'ex_mem_curr->destm'      # Destination M register ID
91 boolsig M_Bch 'ex_mem_curr->takebranch' # Branch Taken flag
92
93 ##### Intermediate Values in Memory Stage ##########################
94 intsig m_valM 'mem_wb_next->valm'       # valM generated by memory
95
96 ##### Pipeline Register W ##########################################
97 intsig W_icode 'mem_wb_curr->icode'     # Instruction code
98 intsig W_dstE 'mem_wb_curr->deste'      # Destination E register ID
99 intsig W_valE  'mem_wb_curr->vale'      # ALU E value
100 intsig W_dstM 'mem_wb_curr->destm'      # Destination M register ID
101 intsig W_valM  'mem_wb_curr->valm'      # Memory M value
102
103 ####################################################################
104 #    Control Signal Definitions.                                   #
105 ####################################################################
106
107 ################ Fetch Stage     ###################################
108
109 ## What address should instruction be fetched at
110 int f_pc = [
111         # Mispredicted branch.  Fetch at incremented PC
112         M_icode == IJXX && !M_Bch : M_valA;
113         # Completion of RET instruction.
114         W_icode == IRET : W_valM;
```

```
115         # Default: Use predicted value of PC
116         1 : F_predPC;
117 ];
118
119 # Does fetched instruction require a regid byte?
120 bool need_regids =
121         f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
122                      IIRMOVL, IRMMOVL, IMRMOVL };
123
124 # Does fetched instruction require a constant word?
125 bool need_valC =
126         f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
127
128 bool instr_valid = f_icode in
129         { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
130                IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
131
132 # Predict next value of PC
133 int new_F_predPC = [
134         f_icode in { IJXX, ICALL } : f_valC;
135         1 : f_valP;
136 ];
137
138
139 ################ Decode Stage ######################################
140
141
142 ## What register should be used as the A source?
143 int new_E_srcA = [
144         D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : D_rA;
145         D_icode in { IPOPL, IRET } : RESP;
146         1 : RNONE; # Don't need register
147 ];
148
149 ## What register should be used as the B source?
150 int new_E_srcB = [
151         D_icode in { IOPL, IRMMOVL, IMRMOVL  } : D_rB;
152         D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
153         1 : RNONE;  # Don't need register
154 ];
155
156 ## What register should be used as the E destination?
157 int new_E_dstE = [
158         D_icode in { IRRMOVL, IIRMOVL, IOPL} : D_rB;
159         D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
160         1 : RNONE;  # Don't need register
161 ];
162
163 ## What register should be used as the M destination?
164 int new_E_dstM = [
```

```
D_icode in { IMRMOVL, IPOPL } : D_rA;
        1 : RNONE;  # Don't need register
];

## What should be the A value?
## Forward into decode stage for valA
int new_E_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        d_srcA == E_dstE : e_valE;    # Forward valE from execute
        d_srcA == M_dstM : m_valM;    # Forward valM from memory
        d_srcA == M_dstE : M_valE;    # Forward valE from memory
        d_srcA == W_dstM : W_valM;    # Forward valM from write back
        d_srcA == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalA;  # Use value read from register file
];

int new_E_valB = [
        d_srcB == E_dstE : e_valE;    # Forward valE from execute
        d_srcB == M_dstM : m_valM;    # Forward valM from memory
        d_srcB == M_dstE : M_valE;    # Forward valE from memory
        d_srcB == W_dstM : W_valM;    # Forward valM from write back
        d_srcB == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalB;  # Use value read from register file
];

################ Execute Stage ######################################

## Select input A to ALU
int aluA = [
        E_icode in { IRRMOVL, IOPL } : E_valA;
        E_icode in { IIRMOVL, IRMMOVL, IMRMOVL } : E_valC;
        E_icode in { ICALL, IPUSHL } : -4;
        E_icode in { IRET, IPOPL } : 4;
        # Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
        E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
                     IPUSHL, IRET, IPOPL } : E_valB;
        E_icode in { IRRMOVL, IIRMOVL } : 0;
        # Other instructions don't need ALU
];

## Set the ALU function
int alufun = [
        E_icode == IOPL : E_ifun;
        1 : ALUADD;
];

```

```
## Should the condition codes be updated?
bool set_cc = E_icode == IOPL;


################ Memory Stage ######################################

## Select memory address
int mem_addr = [
M_icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : M_valE;
        M_icode in { IPOPL, IRET } : M_valA;
        # Other instructions don't need address
];

## Set read control signal
bool mem_read = M_icode in { IMRMOVL, IPOPL, IRET };

## Set write control signal
bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };


################ Pipeline Register Control #########################

# Should I stall or inject a bubble into Pipeline Register F?
# At most one of these can be true.
bool F_bubble = 0;
bool F_stall =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVL, IPOPL } &&
         E_dstM in { d_srcA, d_srcB } ||
        # Stalling at fetch while ret passes through pipeline
        IRET in { D_icode, E_icode, M_icode };

# Should I stall or inject a bubble into Pipeline Register D?
# At most one of these can be true.
bool D_stall =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVL, IPOPL } &&
         E_dstM in { d_srcA, d_srcB };

bool D_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Bch) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        !(E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }) &&
          IRET in { D_icode, E_icode, M_icode };

# Should I stall or inject a bubble into Pipeline Register E?
# At most one of these can be true.
bool E_stall = 0;
```

```
265 bool E_bubble =
266          # Mispredicted branch
267          (E_icode == IJXX && !e_Bch) ||
268          # Conditions for a load/use hazard
269          E_icode in { IMRMOVL, IPOPL } &&
270             E_dstM in { d_srcA, d_srcB};
271
272 # Should I stall or inject a bubble into Pipeline Register M?
273 # At most one of these can be true.
274 bool M_stall = 0;
275 bool M_bubble = 0;
```

code/arch/pipe/pipe-std.hcl

Index