Computer System Organization (CS2.201)

Lecture # 13-19

Processor Architecture and Design / Y86 ISA

Avinash Sharma

Center for Visual Information Technology (CVIT),

IIIT Hyderabad

Slide content acknowledgment: Dr. Suresh Purini & other public sources

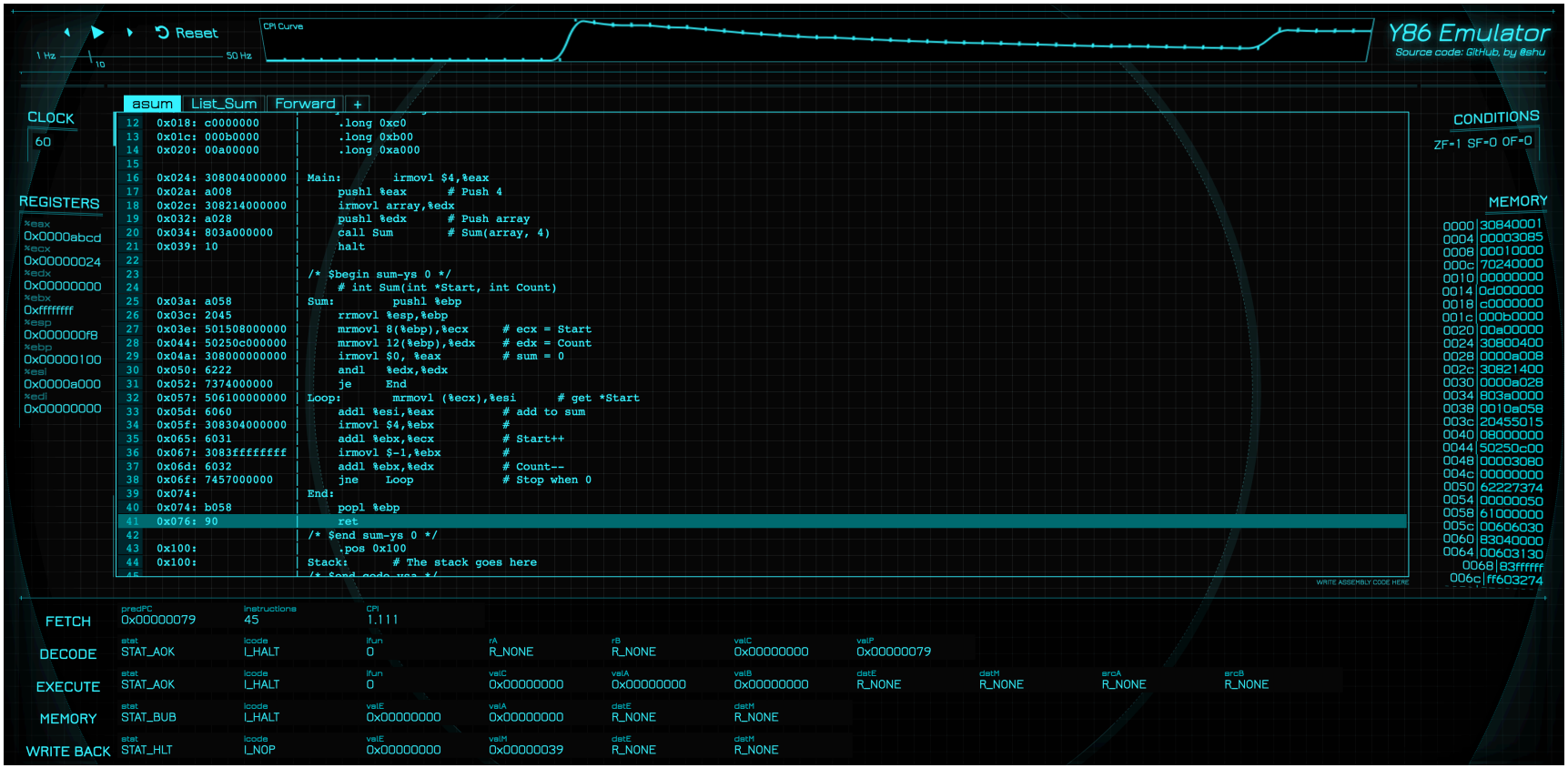
CISC vs RISC

CISC	RISC
Adopted by Intel and other family of processors	Promoted by IBM and later ARM kind of embedded processor designers
Large number of instructions	Fewer instructions (<100)
Allow instructions with long execution times	No instruction with a long execution time
Variable-length instruction encodings	Fixed-length instruction encodings
Multiple formats for specifying operands	Simple addressing formats (base+dispacement)

CISC vs RISC

CISC	RISC
Arithmetic and logical ops can be applied to both memory and register	Arithmetic and logical ops only use register operand
Implementation artifacts hidden from machine- level programs.	Implementation artifacts exposed to machine- level programs.
Condition codes. Special flags are set as a side effect of instructions	No condition codes. Explicit test instructions store the test results in normal registers
Stack-intensive procedure linkage	Register-intensive procedure linkage. Upto 32 registers provisioned.

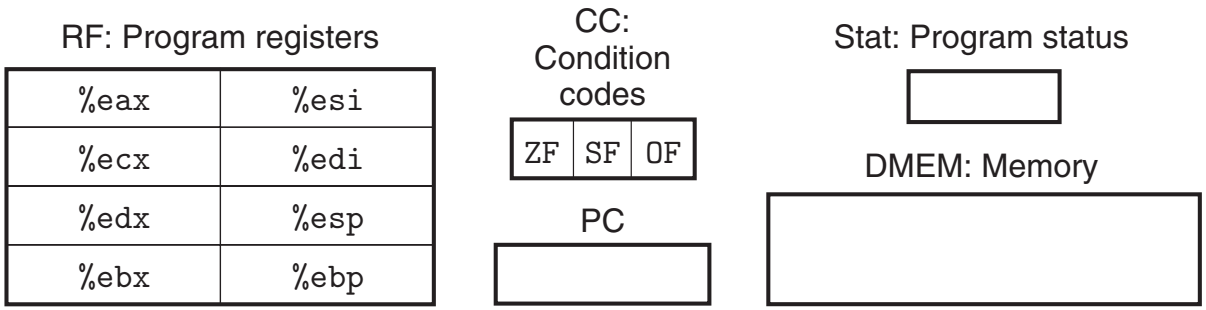
Y86(_64) Simulator/Emulator



Y86_(64) Instruction Set Architecture

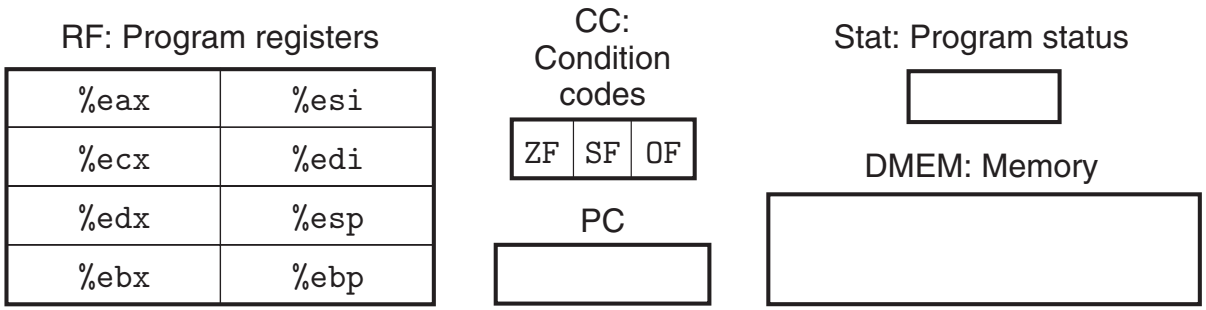
- Y86(_64), is an emulator that can be used for understanding the execution behavior of X86(_64) processors. It can be viewed as taking a CISC instruction set (IA32) and simplifying it by applying some of the principles of RISC
- The Y86(_64) instruction set has fewer data types, instructions, and addressing modes. It also has a simpler byte-level encoding.
- Y86 includes only 4-byte integer operations and includes a smaller set of operations.
- In context of Y86, since we only use 4-byte data, we can refer to these as “words” without any ambiguity.

Y86 Programmer Visible State



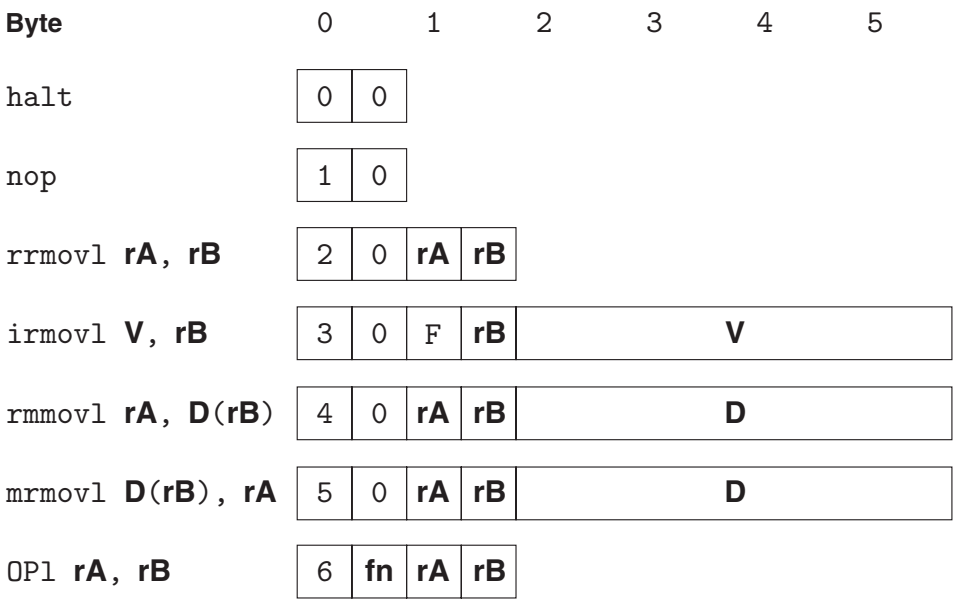
- Only 8 program registers. Each can hold a word **of size 4 bytes**.
- Register %esp holds stack pointer by the push, pop, call, and return instructions.
- Three single-bit condition codes, ZF, SF, and OF, storing information about the effect of the most recent arithmetic or logical instruction.
- The program counter (PC) holds the address of the instruction currently being executed.

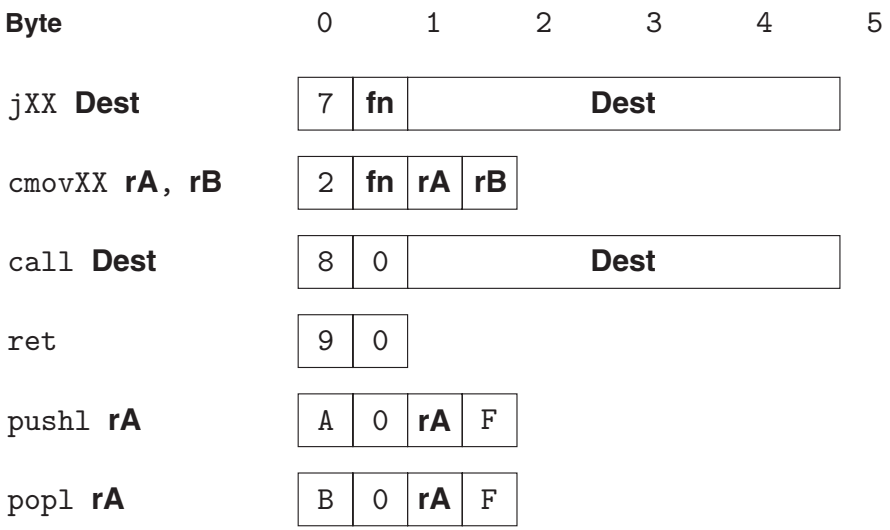
Y86 Programmer Visible State



- 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.
- Status code **Stat**, indicates the overall state of program execution. It will indicate either normal operation, or that some sort of *exception* has occurred.

Y86 Instruction Format: Assembly-code

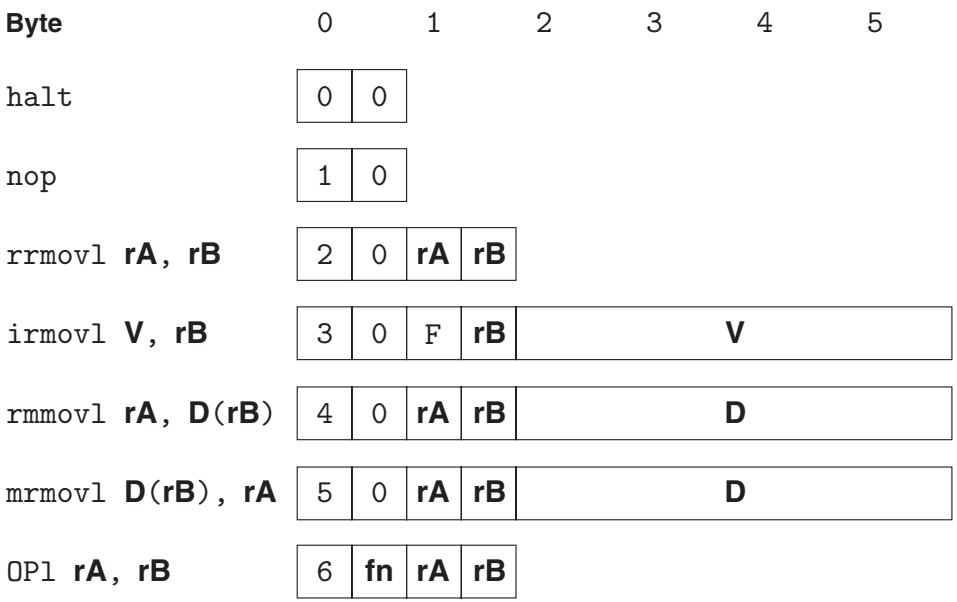


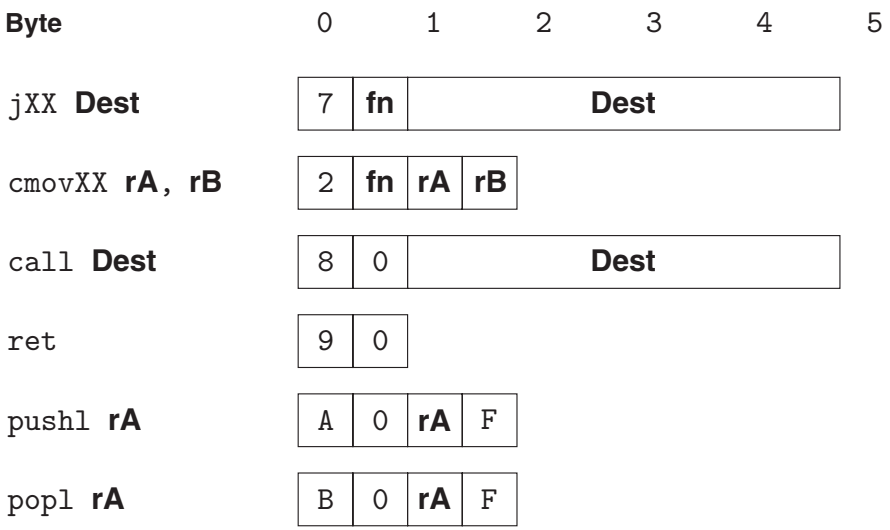


Y86 Instruction Format: Assembly-code

- Some instructions are just 1 byte long, but those that require operands have longer encodings.
- There can be an additional register specifier byte, specifying either one or two registers. These register fields are called rA and rB.
- 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).
- The memory references for the two memory movement instructions have a simple base and displacement format.
- NO support for the second index register or any scaling of a register's value in the address computation.

Y86 Instruction Format: Assembly-code



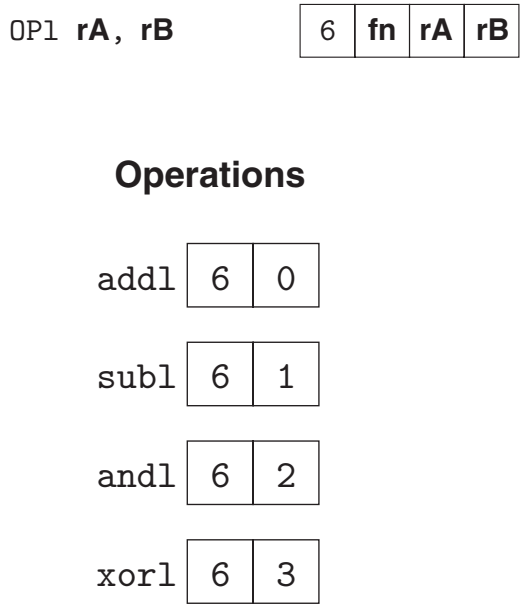


Y86 Instruction Format: Assembly-code

- The halt instruction stops instruction execution, with the status code set to HLT. IA32 has a comparable instruction, called hlt.
- NOP causes idle state for the processor, useful in clock synchronization cases.
- 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.

Y86 Instruction Format: Assembly-code

- There are four integer operation instructions, shown 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).



Y86 Instruction Format: Assembly-code

`jXX` **Dest**

- The seven jump instructions (jXX), namely, 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

Branches

jmp	7	0	jne	7	4
jle	7	1	jge	7	5
jl	7	2	jg	7	6
je	7	3			

Y86 Instruction Format: Assembly-code

- There are six conditional move instructions (shown as cmovXX): cmovle, cmovl, cmove, cmovne, cmovge, and cmovg.
- These have the same format as the register-register move instruction rrmovl, but the destination register is updated only if the condition codes satisfy the required constraints.

Moves

rrmovl	2	0	cmovne	2	4
cmovle	2	1	cmovge	2	5
cmovl	2	2	cmovg	2	6
cmove	2	3			

Y86 Instruction Encodings

- 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.

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

Y86 States/Exceptions

Value	Name	Meaning
1	`AOK`	Normal operation
2	`HLT`	`halt` instruction encountered
3	`ADR`	Invalid address encountered
4	`INS`	Invalid instruction encountered

- Y86 simply have the processor stop executing instructions when it encounters any of the exceptions listed.
- In a more complete design, the processor would typically invoke an exception handler, a procedure designated to handle the specific type of exception encountered.

X86 vs Y86 Programs

- The Y86 code is essentially the same as IA32, except that Y86 sometimes requires two instructions to accomplish what can be done with a single IA32 instruction (e.g., memory operands are not allowed in Y86).
- 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.
- It follows many of the programming conventions we have seen for IA32, including the use of the stack and frame pointers.
- For simplicity, it does not follow the IA32 convention of having some registers designated as callee-save registers.

Y86 Programs

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

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

Y86 Programs

```
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           call Main               # Execute main program
6           halt                    # Terminate program
7
8   # Array of 4 elements
9           .align 4
10  array:  .long 0xd
11          .long 0xc0
12          .long 0xb00
13          .long 0xa000
14
15  Main:   pushl %ebp
16          rrmovl %esp,%ebp
17          irmovl $4,%eax
18          pushl %eax              # Push 4
19          irmovl array,%edx
20          pushl %edx              # Push array
21          call Sum                # Sum(array, 4)
22          rrmovl %ebp,%esp
23          popl %ebp
24          ret
25
26          # int Sum(int *Start, int Count)
27  Sum:    pushl %ebp
28          rrmovl %esp,%ebp
29          mrmovl 8(%ebp),%ecx     # ecx = Start
30          mrmovl 12(%ebp),%edx    # edx = Count
31          xorl %eax,%eax          # sum = 0
32          andl   %edx,%edx        # Set condition codes
33          je     End
34  Loop:   mrmovl (%ecx),%esi      # get *Start
35          addl %esi,%eax          # add to sum
36          irmovl $4,%ebx          #
37          addl %ebx,%ecx          # Start++
38          irmovl $-1,%ebx         #
39          addl %ebx,%edx          # Count--
40          jne    Loop             # Stop when 0
41  End:    rrmovl %ebp,%esp
42          popl %ebp
43          ret
44
45  # The stack starts here and grows to lower addresses
46          .pos 0x100
47  Stack:
```

Y86 Programs

- 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 address for 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 47), to indicate address 0x100 using a .pos directive (line 46).
- Our stack will therefore start at this address and grow toward lower addresses. We must ensure that the stack does not grow so large that it overwrites the code or other program data.

Y86 Programs

- Lines 9 to 13 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 4-byte boundary (using the .align directive).
- Lines 17 to 6 show a “main” procedure that calls the function Sum on the four-word array and then halts.
-

Sequential Y86 Implementations

- The speed of a computer processor, or CPU, is determined by the clock cycle, which is the amount of time between two pulses of an oscillator.
- Typically, the higher number of pulses per second, the faster the computer processor can to process information. For example, a 4 GHz processor performs 4,000,000,000 clock cycles per second.
- Computer processors can execute one or more instructions per clock cycle, depending on the type of processor.

Sequential Y86 Implementations

- As a first step, let's 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.
- The purpose of developing SEQ is to provide a first step toward our ultimate goal of implementing an efficient, pipelined processor.

Sequential Y86 Instruction Stages

Each instruction sequentially go through following common stages:

1. **Fetch**
2. **Decode**
3. **Execute**
4. **Memory**
5. **Write-back**
6. **PC update**

The processor loops indefinitely, performing these stages unless any exception condition occurs.

Sequential Y86 Instruction Stages

Why common stages for all instructions ?

- Using a very simple and uniform structure is important when designing hardware, since we want to minimize the total amount of hardware which is ultimately map it onto the 2D integrated-circuit chip.
- One way to minimize the complexity is to have the different instructions share as much of the hardware as possible, e.g., single ALU unit in a processor used for all instructions in different ways.
- The cost of duplicating blocks of logic in hardware is much higher than the cost of having multiple copies of code in software.

Sequential Y86 Instruction Stages

Fetch:

- The fetch stage reads the bytes of an instruction from memory, using the PC as the memory address. It extracts the two 4-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 4-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.

Sequential Y86 Instruction Stages

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.

Sequential Y86 Instruction Stages

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.

Sequential Y86 Instruction Stages

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.

Sequential Y86 Instruction Stages

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

Sequential Y86 Instruction Stages

Stage	`OPl` rA, rB	`rrmovl` rA, rB
Fetch	icode : ifun $\leftarrow$ M_1[PC] rA : rB $\leftarrow$ M_1[PC + 1]	icode : ifun $\leftarrow$ M_1[PC] rA : rB $\leftarrow$ M_1[PC + 1]
	valP $\leftarrow$ PC + 2	valP $\leftarrow$ PC + 2
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
Memory		
Write back	R[rB] $\leftarrow$ valE	R[rB] $\leftarrow$ valE
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP

Sequential Y86 Instruction 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

Sequential Y86 Instruction Stages

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

Sequential Y86 Instruction Stages

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

Sequential Y86 Instruction Stages

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

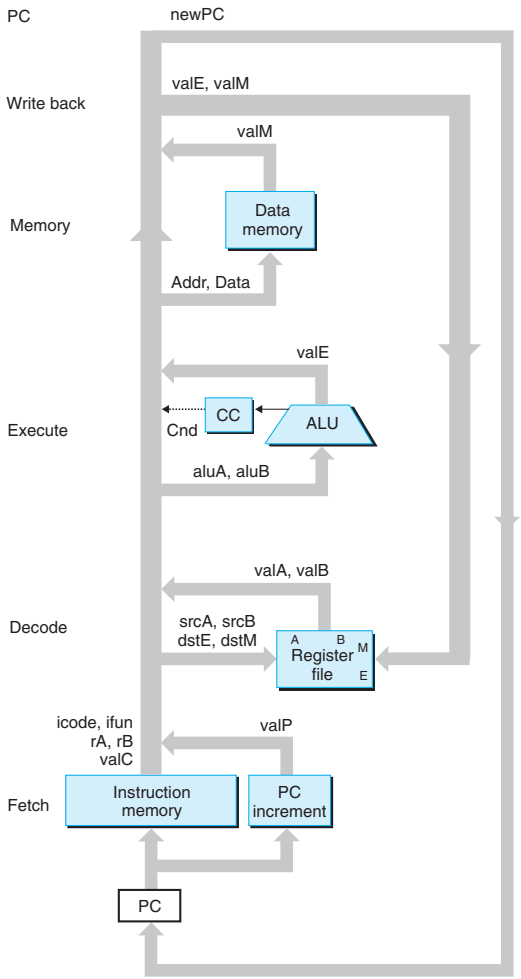
SEQ Y86 Hardware Structure

Fetch:

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

Decode:

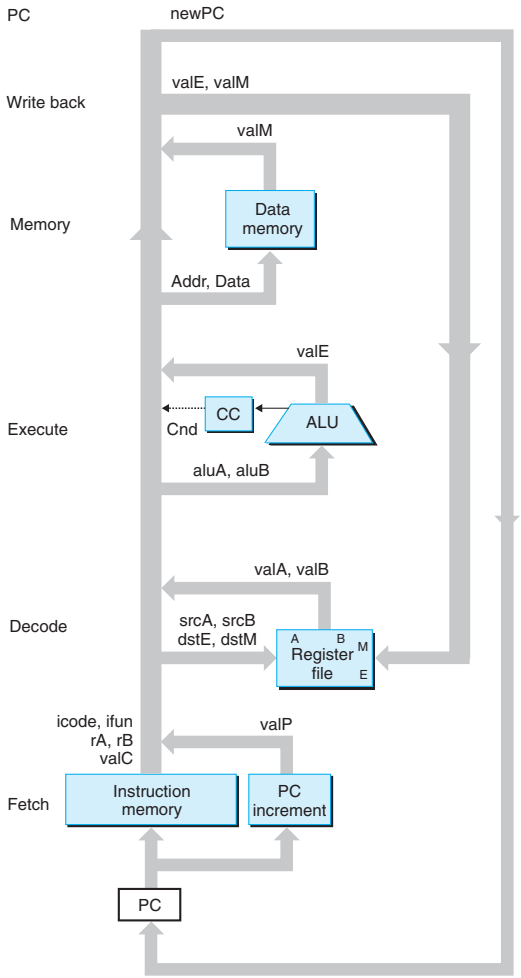
- The register file has two read ports, A and B, via which register values valA and valB are read simultaneously.



SEQ Y86 Hardware Structure

Execute:

- The execute stage uses the (ALU) unit for different purposes according to the instruction type.
- 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 zero.
- The CC holds the three condition-code bits. New values for the condition codes are set by the ALU. In a jump instruction, the branch signal Cnd is computed based on the condition codes and the jump type.



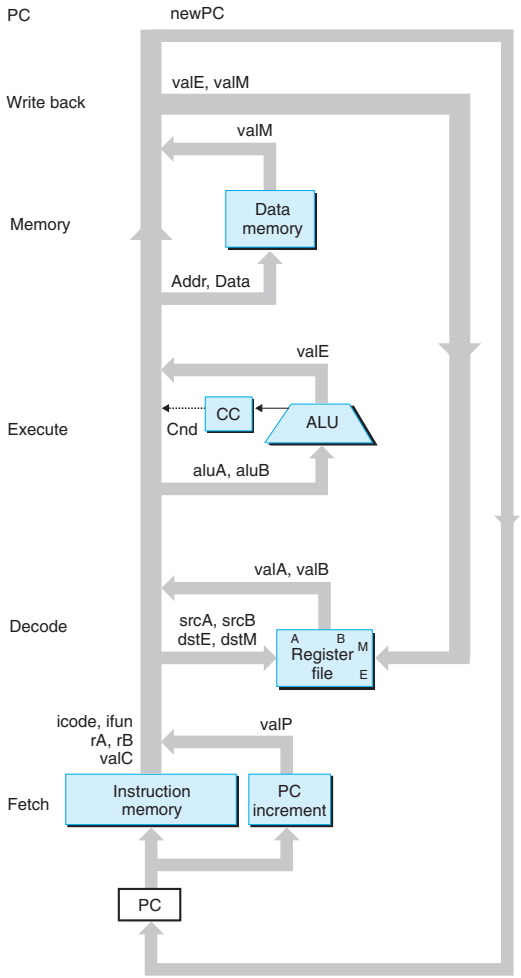
SEQ Y86 Hardware Structure

Memory:

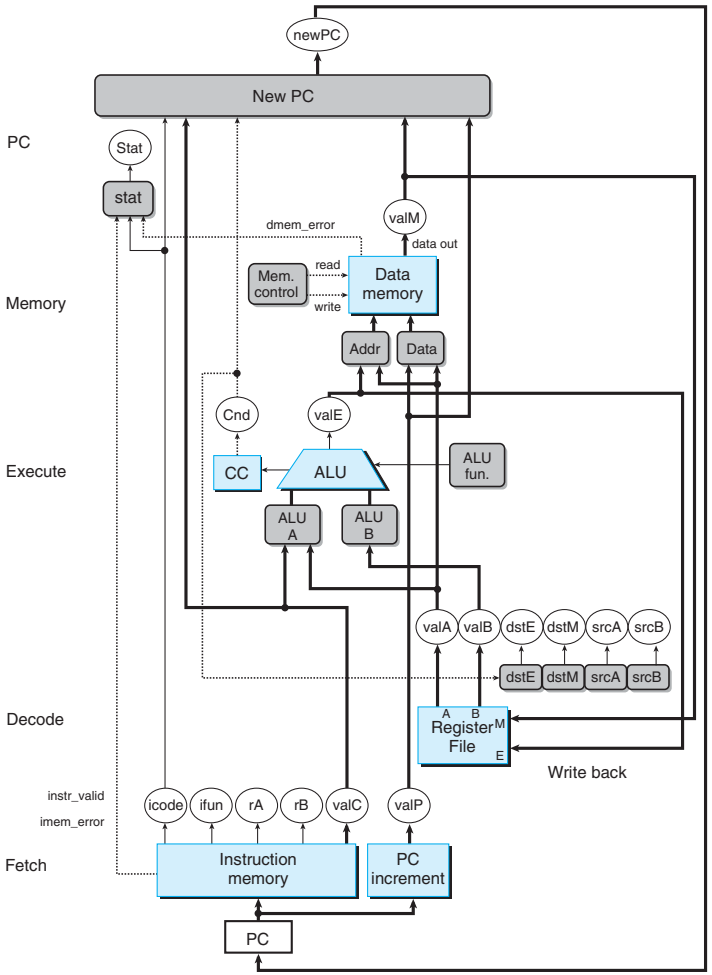
- The data memory reads or writes a word of memory when executing a memory instruction. The instruction and data memories access the same memory locations, but for different purposes.

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.

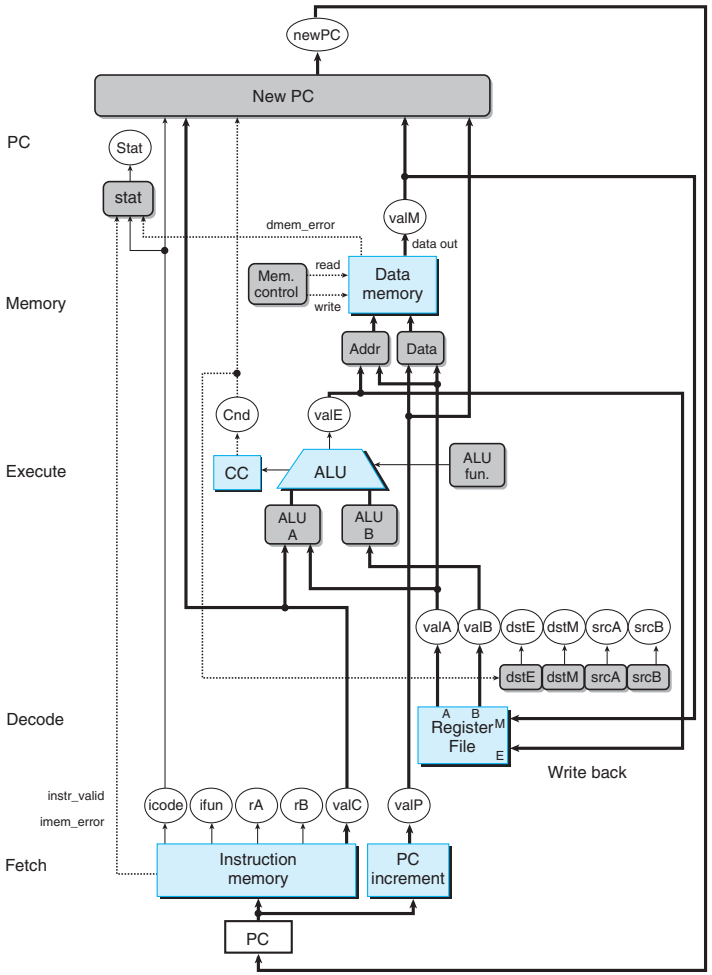


SEQ Y86 Hardware Structure



- Hardware units are shown as light blue boxes.
- Control logic blocks are drawn as gray rounded rectangles.
- Wire names are indicated in white round boxes.
- Word-wide data connections are shown as medium lines
- Byte and narrower data connections are shown as thin lines
- Single-bit connections are shown as dotted lines.

SEQ Y86 Hardware Structure



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

SEQ Y86 Timing

SEQ Timing:

- The hardware structure operates with a single clock transition triggering a flow through combinational logic to execute an entire instruction.
- This 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.

SEQ Y86 Timing

SEQ Timing:

- We can also assume that reading from a random- access memory operates much like combinational logic, with the output word generated based on the address input.
- This is a reasonable assumption for smaller memories and we can mimic this effect for larger circuits using special clock circuits. 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.**

SEQ Y86 Timing

SEQ Timing:

- 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 0xF as a port address to indicate that no write should be performed for this port.

SEQ Y86 Timing

SEQ Timing:

- This clocking of the registers and memories is all that is required to control the sequencing of activities in our processor.
- Y86 SEQ hardware achieves the same effect as would a sequential execution of the assignments, 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.

SEQ Y86 Timing

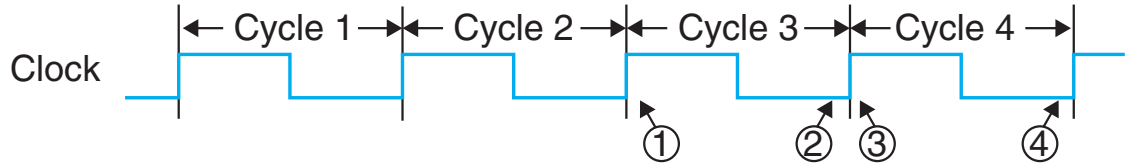
SEQ Timing:

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

`pushl` rA
icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC + 1]
valP $\leftarrow$ PC + 2
valA $\leftarrow$ R[rA] valB $\leftarrow$ R[%esp]
valE $\leftarrow$ valB + (−4)
M_4[valE] $\leftarrow$ valA
R[%esp] $\leftarrow$ valE
PC $\leftarrow$ valP

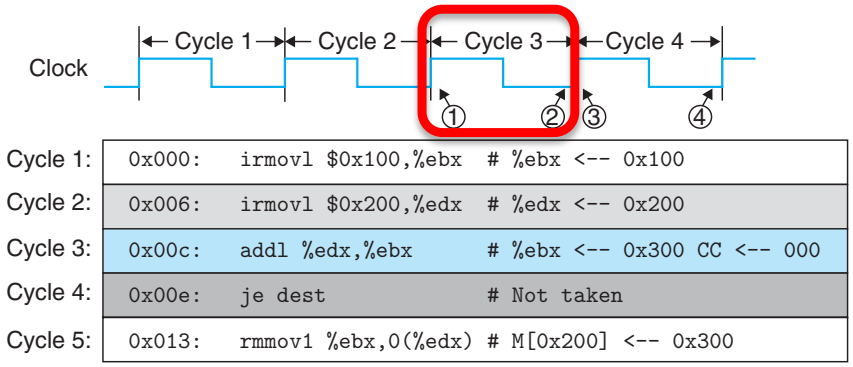
SEQ Y86 Timing

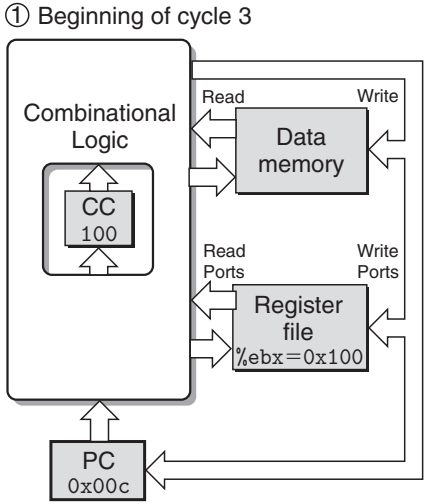
```
1       0x000:   irmovl $0x100,%ebx  # %ebx <-- 0x100
2       0x006:   irmovl $0x200,%edx  # %edx <-- 0x200
3       0x00c:   addl %edx,%ebx      # %ebx <-- 0x300 CC <-- 000
4       0x00e:   je dest             # Not taken
5       0x013:   rmmovl %ebx,0(%edx) # M[0x200] <-- 0x300
6       0x019: dest: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

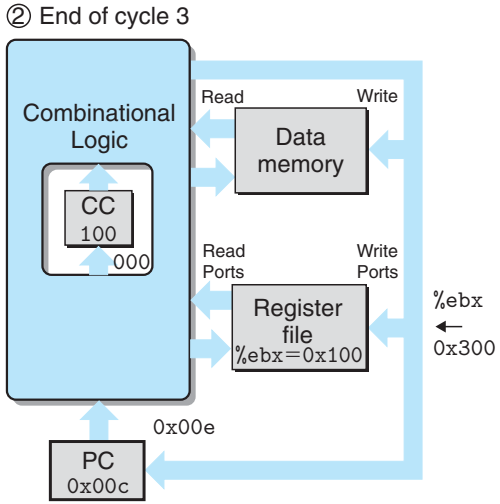


Cycle 1:	0x000:	irmovl $0x100,%ebx	# %ebx <-- 0x100
Cycle 2:	0x006:	irmovl $0x200,%edx	# %edx <-- 0x200
Cycle 3:	0x00c:	addl %edx,%ebx	# %ebx <-- 0x300 CC <-- 000
Cycle 4:	0x00e:	je dest	# Not taken
Cycle 5:	0x013:	rmmov1 %ebx,0(%edx)	# M[0x200] <-- 0x300

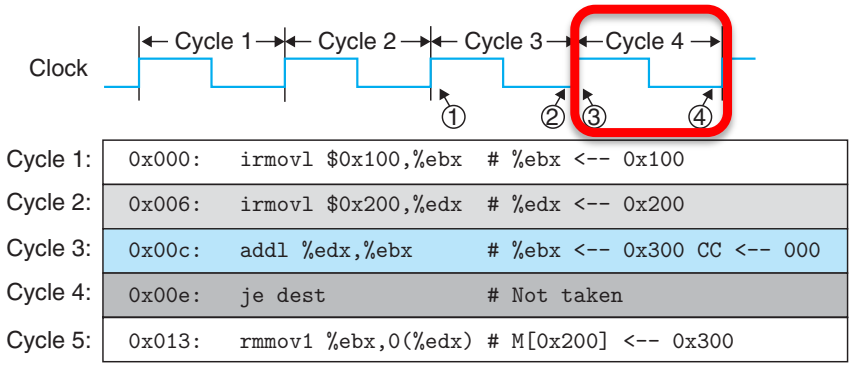
SEQ Y86 Timing



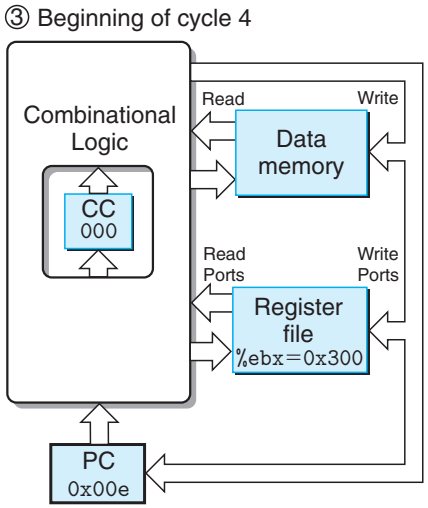


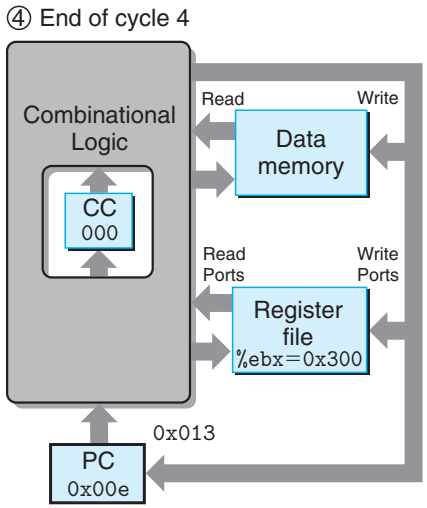


SEQ Y86 Timing



Cycle 1:	0x000:	irmovl $0x100,%ebx	# %ebx <-- 0x100
Cycle 2:	0x006:	irmovl $0x200,%edx	# %edx <-- 0x200
Cycle 3:	0x00c:	addl %edx,%ebx	# %ebx <-- 0x300 CC <-- 000
Cycle 4:	0x00e:	je dest	# Not taken
Cycle 5:	0x013:	rmmov1 %ebx,0(%edx)	# M[0x200] <-- 0x300

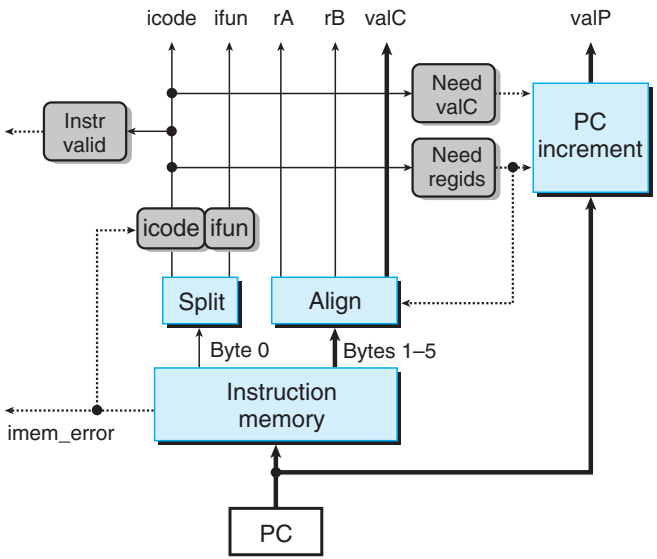




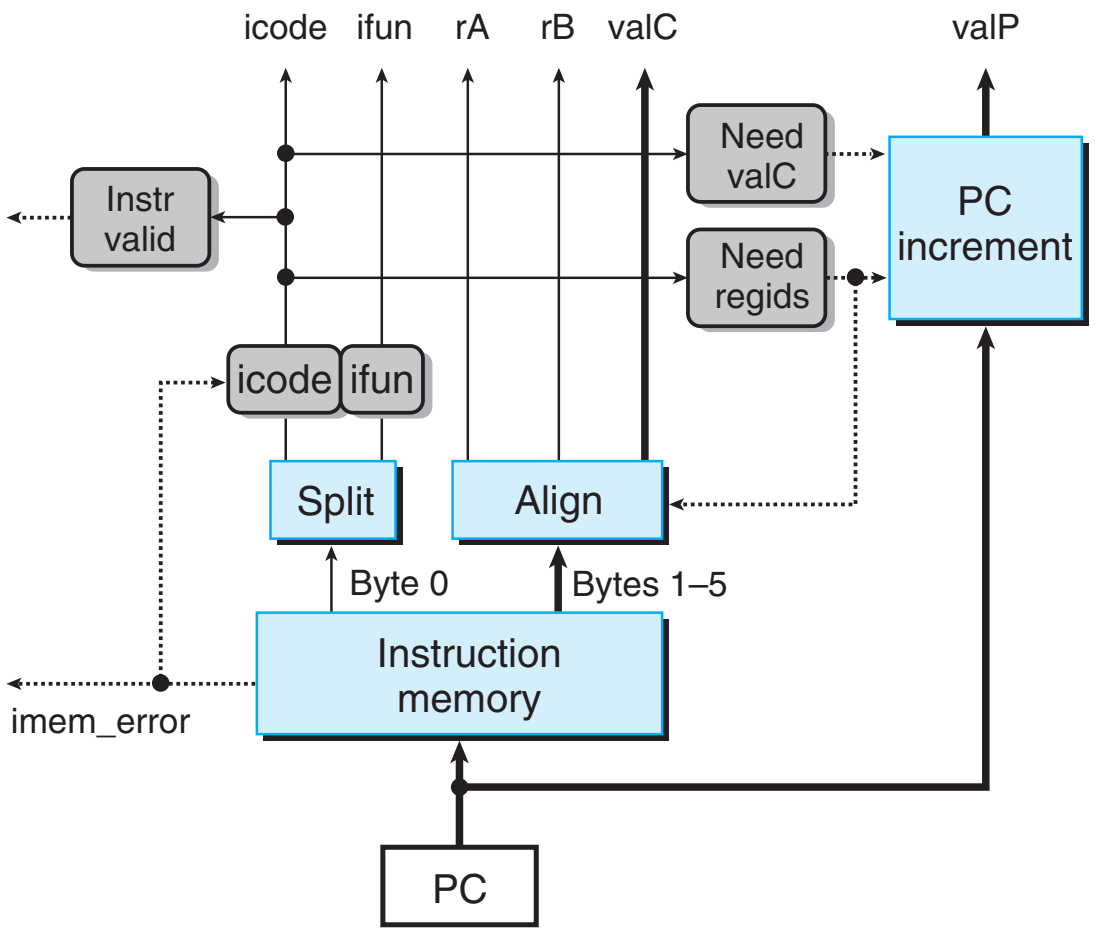
SEQ Y86 Stage Implementations

Fetch Stage:

- The fetch stage includes the instruction memory hardware unit.
- This unit reads 6 bytes from memory at a time, using the PC as the address of the first byte (byte 0).
- The control logic blocks labeled “icode” and “ifun” then compute the instruction and function codes.



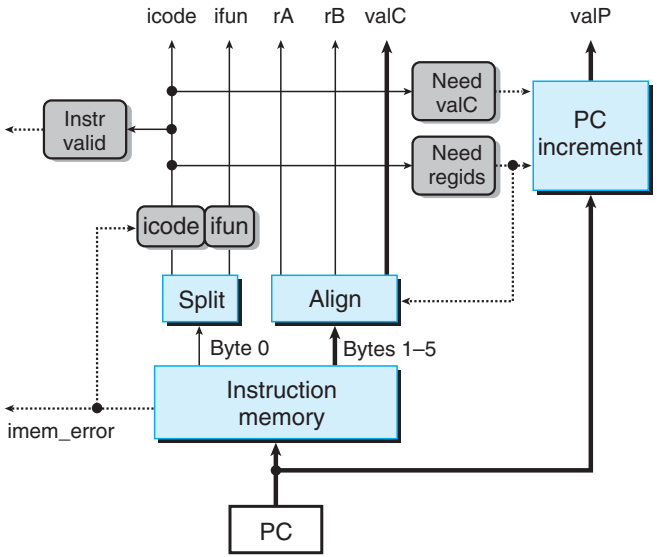
SEQ Y86 Stage Implementations



SEQ Y86 Stage Implementations

Fetch Stage:

- If the values read from memory is not a valid instruction or, in the event that the instruction address is not valid (as indicated by the signal imem_error), the values corresponding to a nop instruction.
- Based on the value of icode, we can compute three 1-bit signals.



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

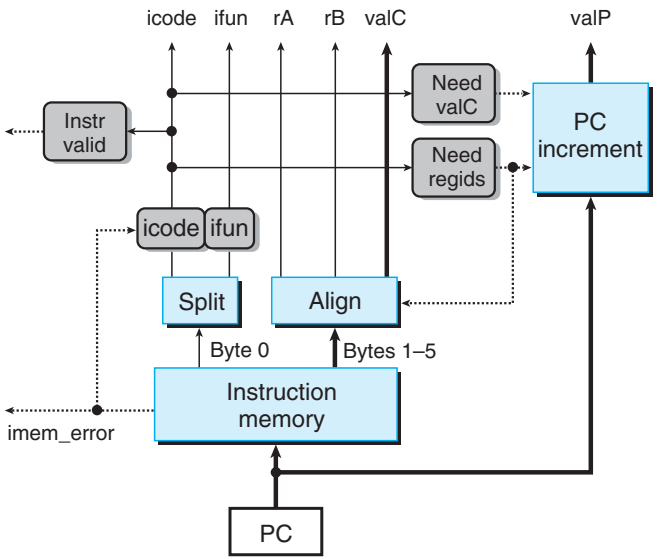
need_regids: Does this instruction include a register specifier byte?

need_valC: Does this instruction include a constant word?

SEQ Y86 Stage Implementations

Fetch Stage:

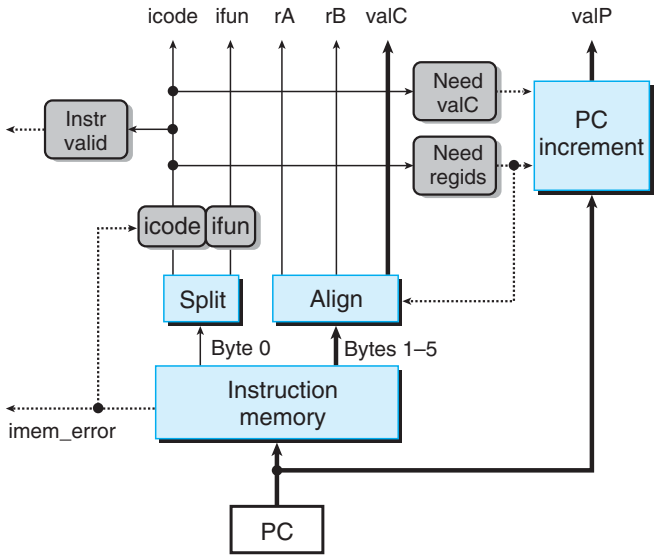
- The remaining 5 bytes read from the instruction memory encode some combination of the register specifier byte and the constant word.
- These bytes are processed by the hardware unit labeled “Align” into the register fields and the constant word.
- When the computed signal need_regids is 1, then byte 1 is split into register specifiers rA and rB. Otherwise, these two fields are set to 0xF (RNONE).



SEQ Y86 Stage Implementations

Fetch Stage:

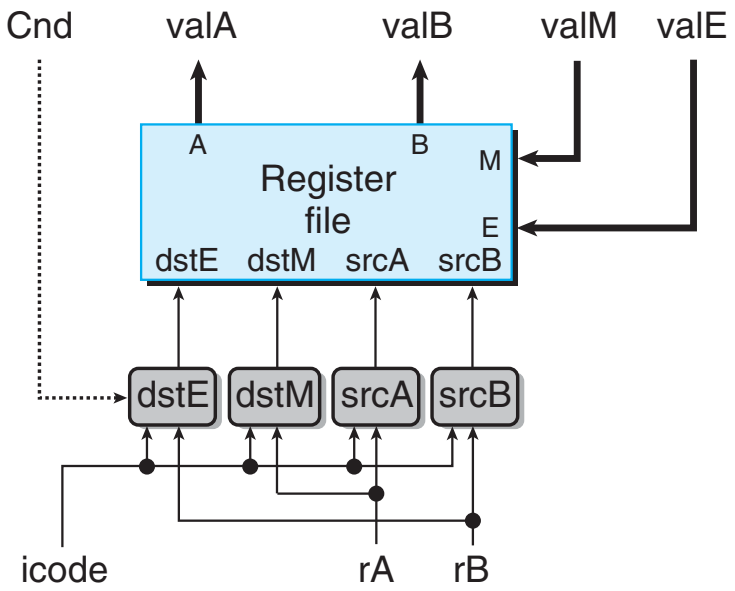
- The unit labeled “Align” also generates the constant word valC. This will either be bytes 1 to 4 or bytes 2 to 5, depending on the value of signal need_regids.
- The PC incrementer hardware unit generates the signal valP, based on the current value of the PC, and the two signals need_regids and need_valC. For PC value p, need_regids value r, and need_valC value i, the incrementer generates the value p + 1 + r + 4i.



SEQ Y86 Stage Implementations

Decode & Write-back Stages:

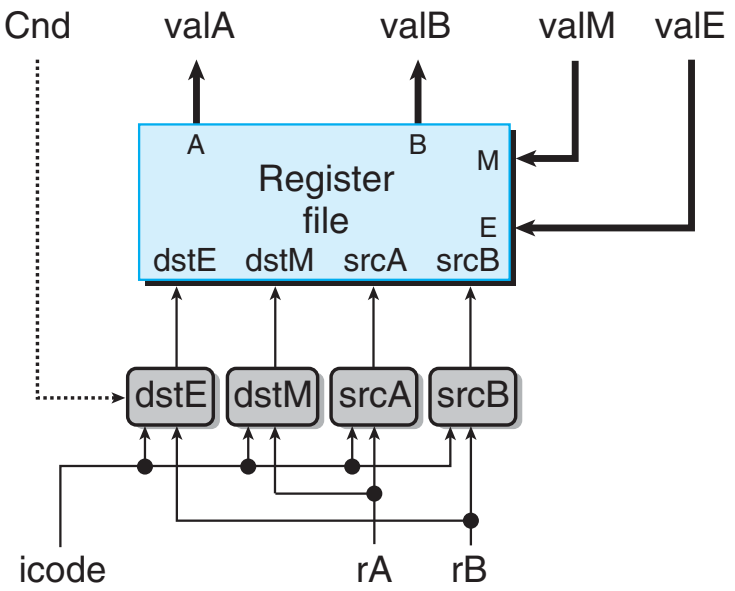
- These two stages are combined because they both access the register file.
- The register file has four ports. It supports up to two simultaneous reads (on ports A and B) and two simultaneous writes (on ports E and M).
- Each port has both an address connection and a data connection, where the address connection is a register ID (4 wires).
- 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.



SEQ Y86 Stage Implementations

Decode & Write-back Stages:

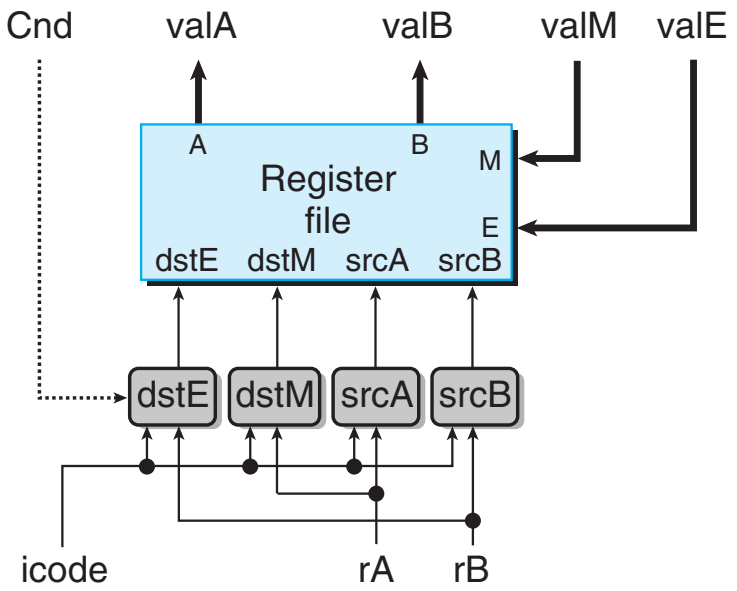
- The two read ports have address inputs srcA and srcB, while the two write ports have address inputs dstE and dstM.
- The special identifier 0xF (RNONE) on an address port indicates that no register should be accessed.
- The four blocks at the bottom generate the four different register IDs for the register file, based on the instruction code icode, the register specifiers rA and rB, and possibly the condition signal Cnd computed in the execute stage.



SEQ Y86 Stage Implementations

Decode & Write-back Stages:

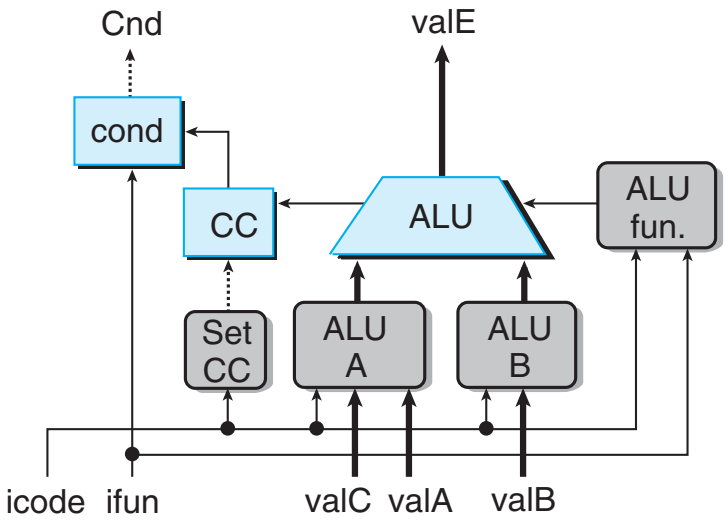
- Register ID srcA indicates which register should be read to generate valA. The desired value depends on the instruction type.
- Register ID dstE indicates the destination register for write port E, where the computed value valE is stored.



SEQ Y86 Stage Implementations

Execute Stage:

- The execute stage includes the arithmetic/logic unit (ALU). The data and control signals are generated by three control blocks. The ALU output becomes the signal valE.
- 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.

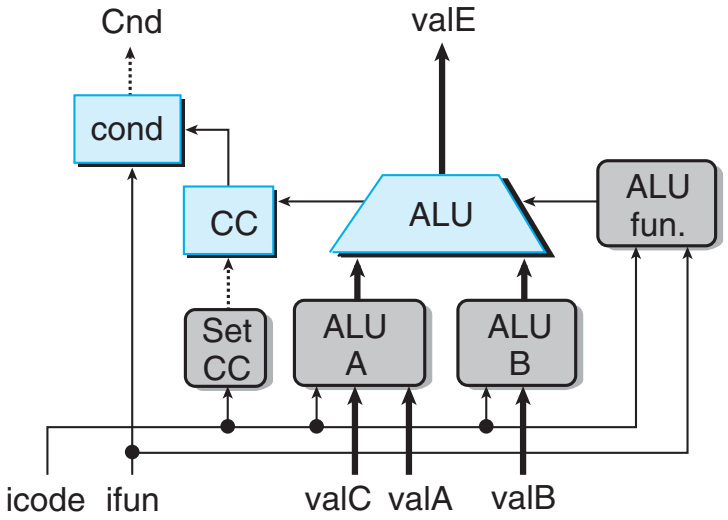


SEQ Y86 Stage Implementations

Execute Stage:

- 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. Specific icode generate a signal Set_CC that controls whether or not the condition code register should be updated

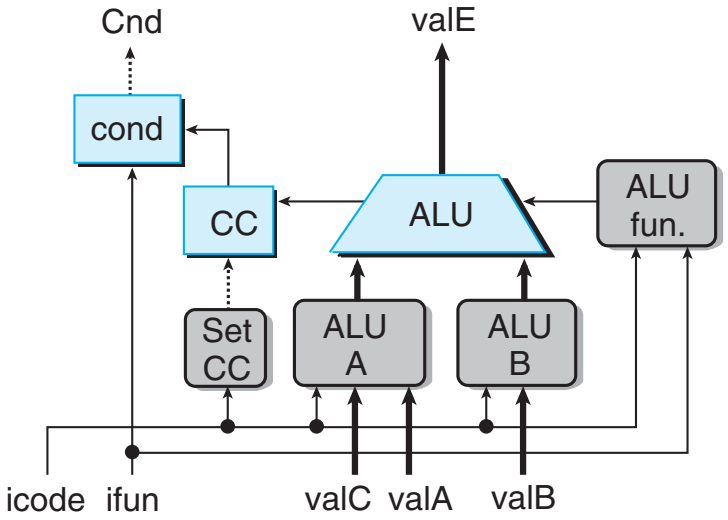
.



SEQ Y86 Stage Implementations

Execute Stage:

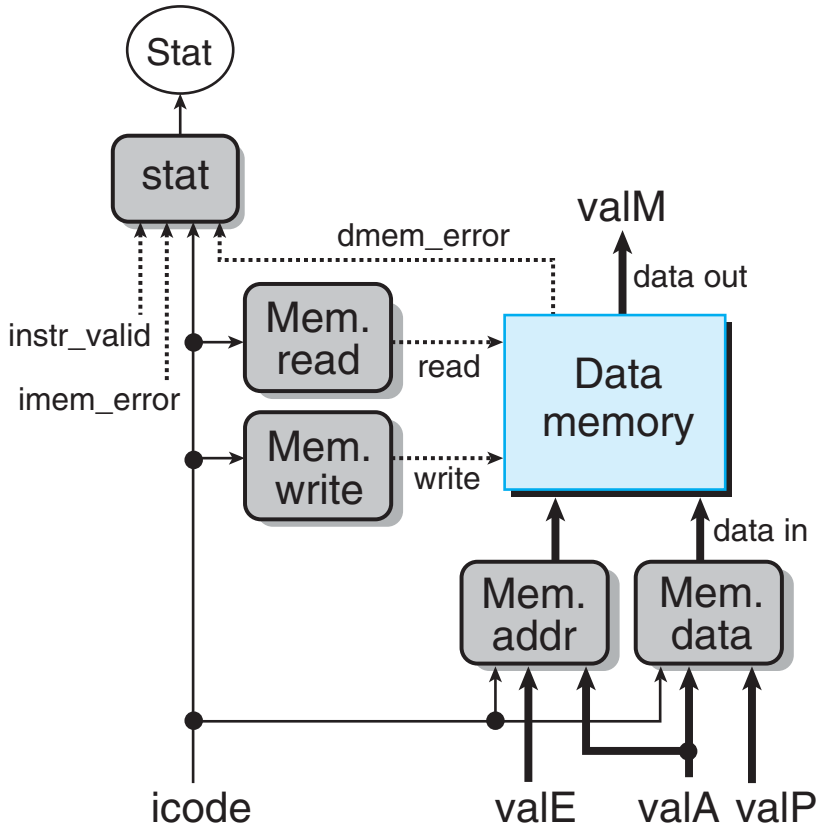
- The hardware unit labeled “cond” uses a combination of the condition codes and the function code to determine whether a conditional branch or data transfer should take place.
- It generates the Cnd signal used both for the setting of dstE with conditional moves, and in the next PC logic for conditional branches.
- For other instructions, it will be ignored by the control logic.



SEQ Y86 Stage Implementations

Memory Stage:

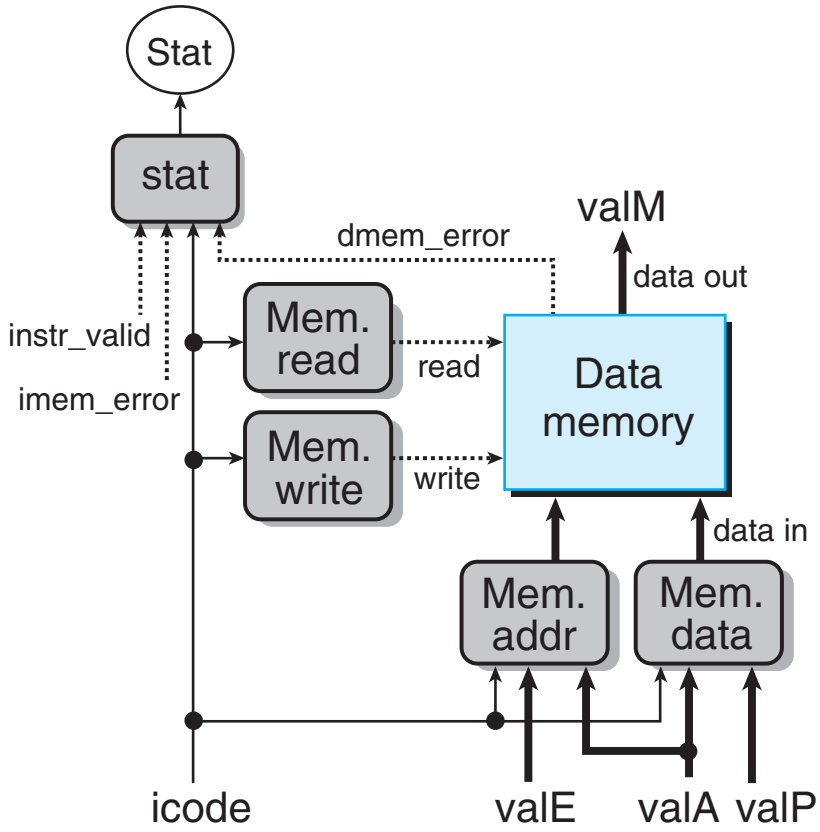
- The memory stage has the task of either reading or writing program data.
- Two control blocks generate the values for the memory address and the memory input data (for write operations).
- Two other blocks generate the control signals indicating whether to perform a read or a write operation.
- When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



SEQ Y86 Stage Implementations

Memory Stage:

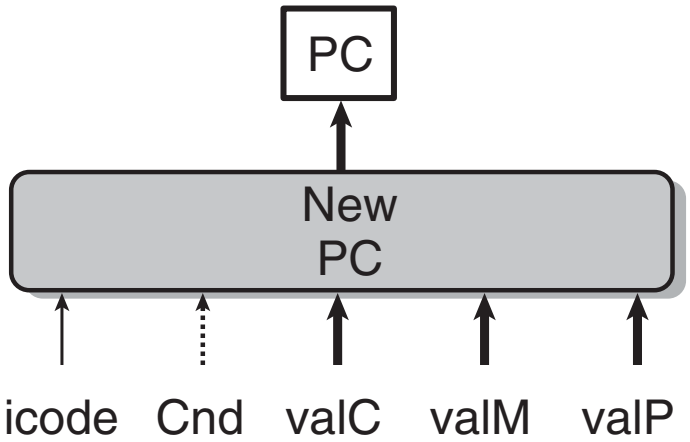
- A final function for the memory stage is to compute the status code Stat resulting from the instruction execution, according to the values of *icode*, *imem_error, instr_valid* generated in the fetch stage, and the signal *dmem_error* generated by the data memory.



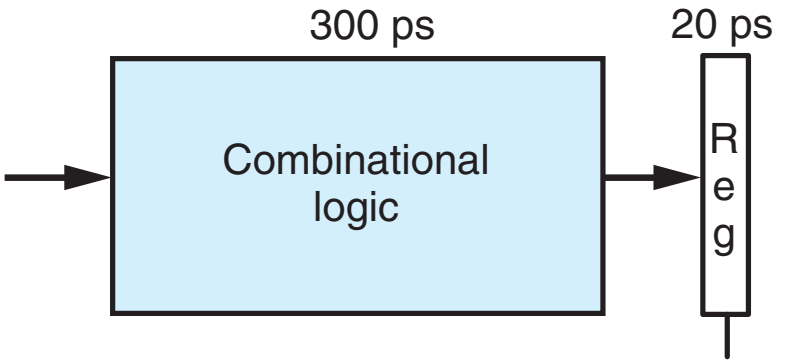
SEQ Y86 Stage Implementations

PC Update Stage:

- The final stage in SEQ generates the new value of the program counter.
- The new PC will be valC, valM, or valP, depending on the instruction type and whether or not a branch should be taken.



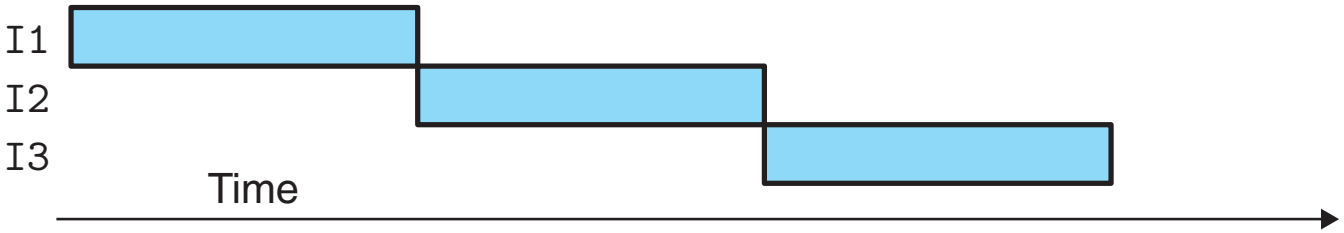
Pipelined Instruction Processing



Delay = 320 ps
Throughput = 3.12 GIPS

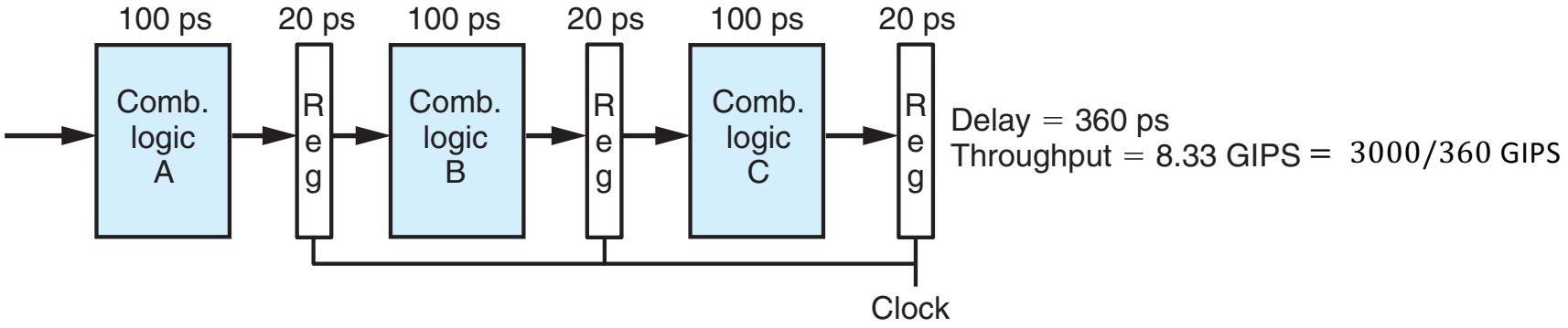
$$\text{Throughput} = \frac{1\ \text{instruction}}{(20+300)\ \text{picosecond}} \cdot \frac{1000\ \text{picosecond}}{1\ \text{nanosecond}} \approx 3.12\ \text{GIPS}$$

(a) Hardware: Unpipelined

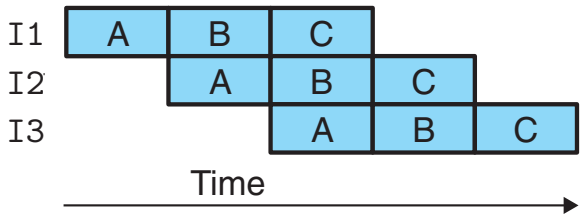


(b) Pipeline diagram

Pipelined Instruction Processing

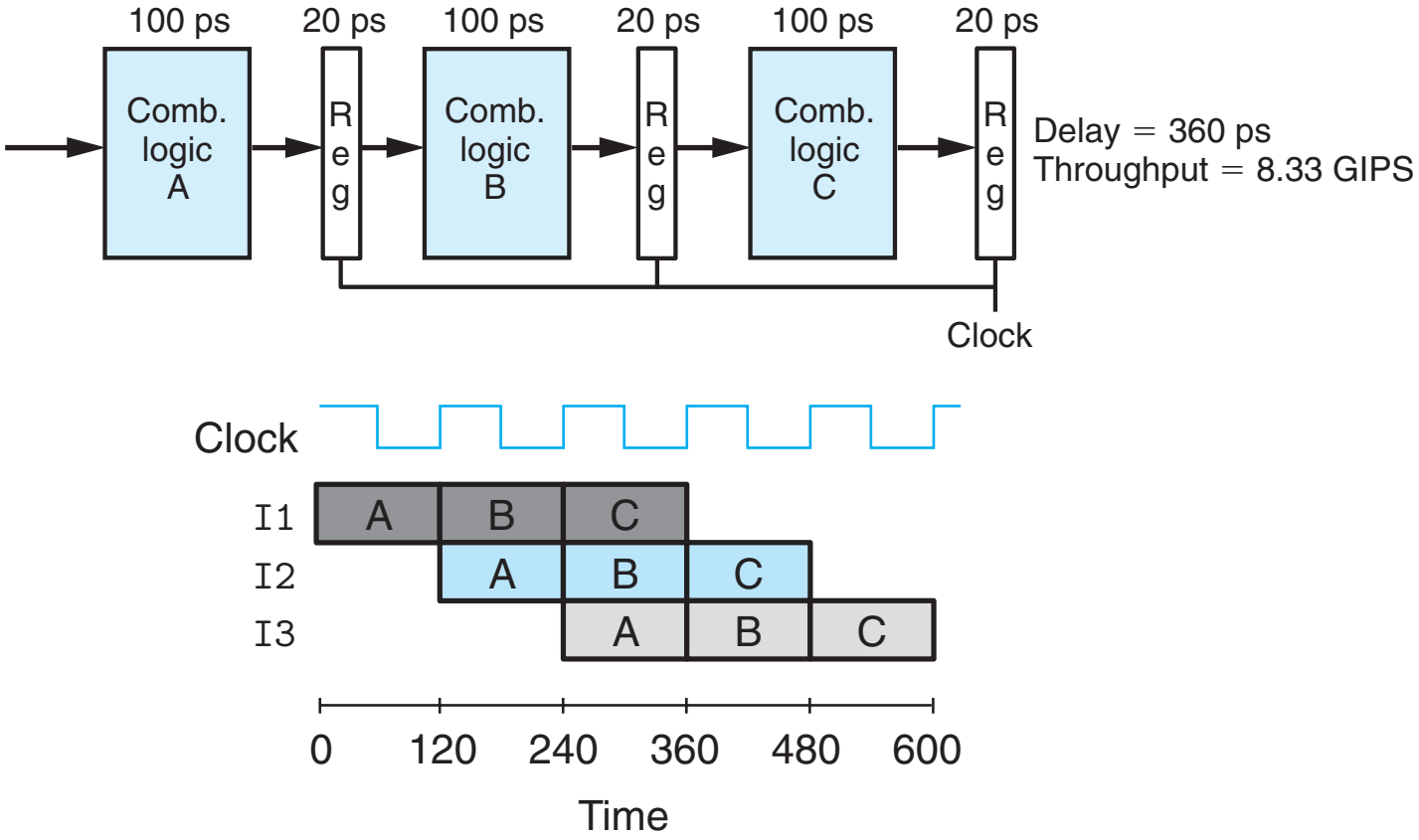


(a) Hardware: Three-stage pipeline

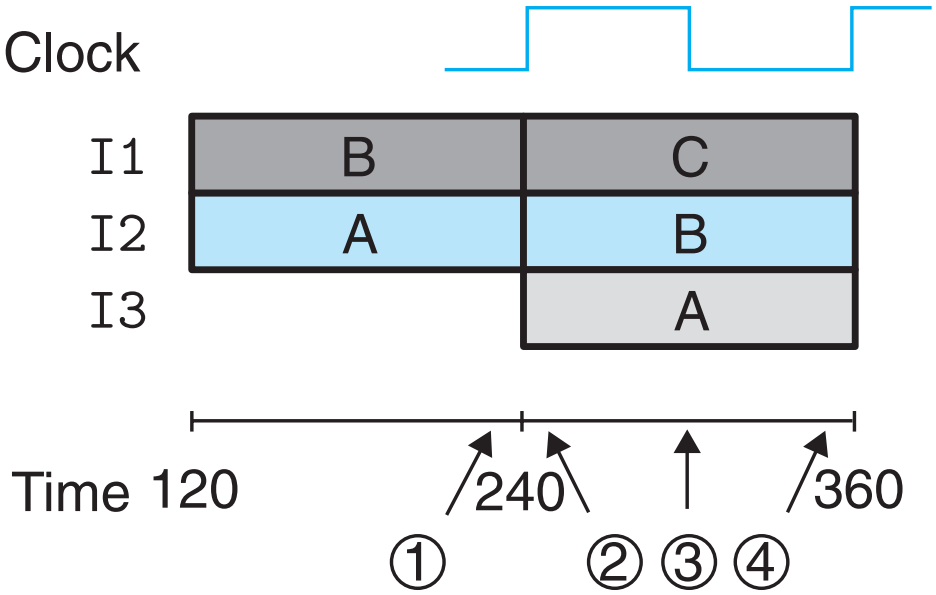


(b) Pipeline diagram

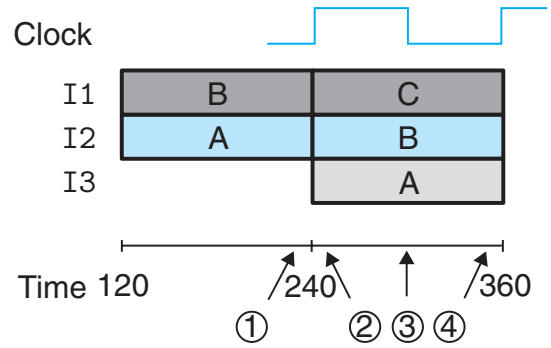
Pipelined Instruction Processing



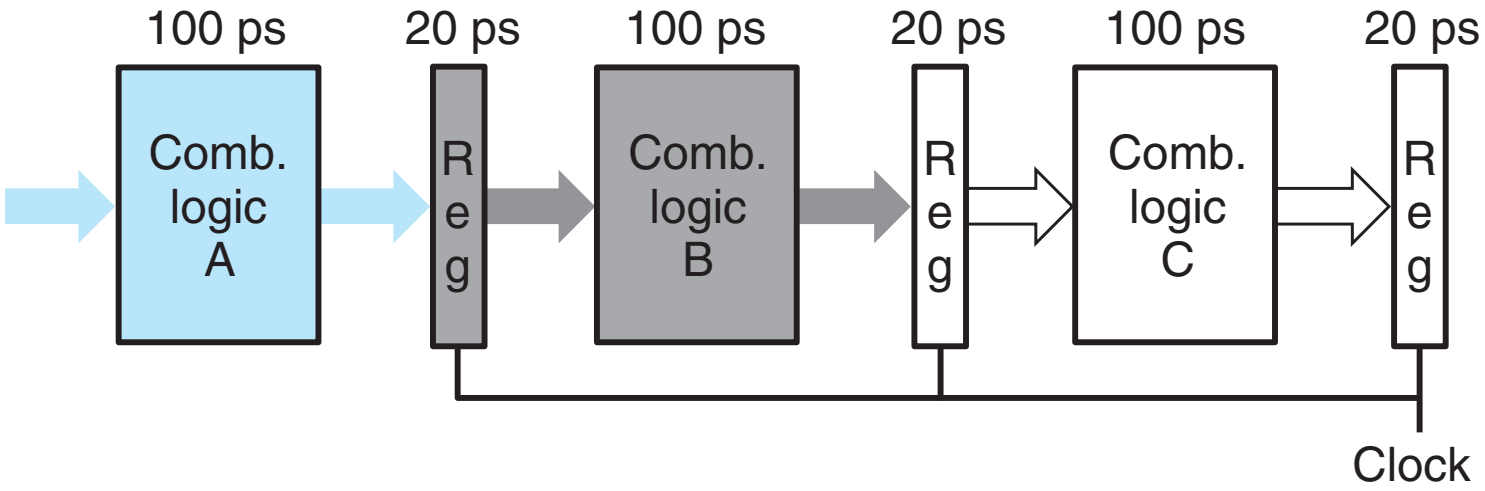
Pipelined Instruction Processing



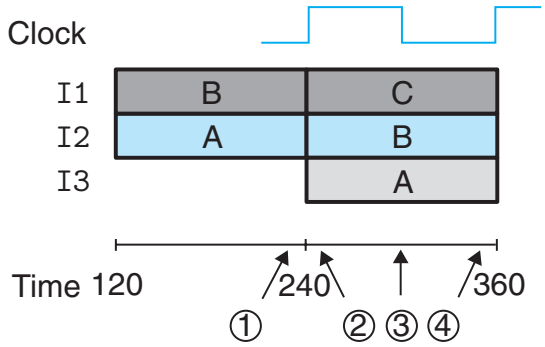
Pipelined Instruction Processing



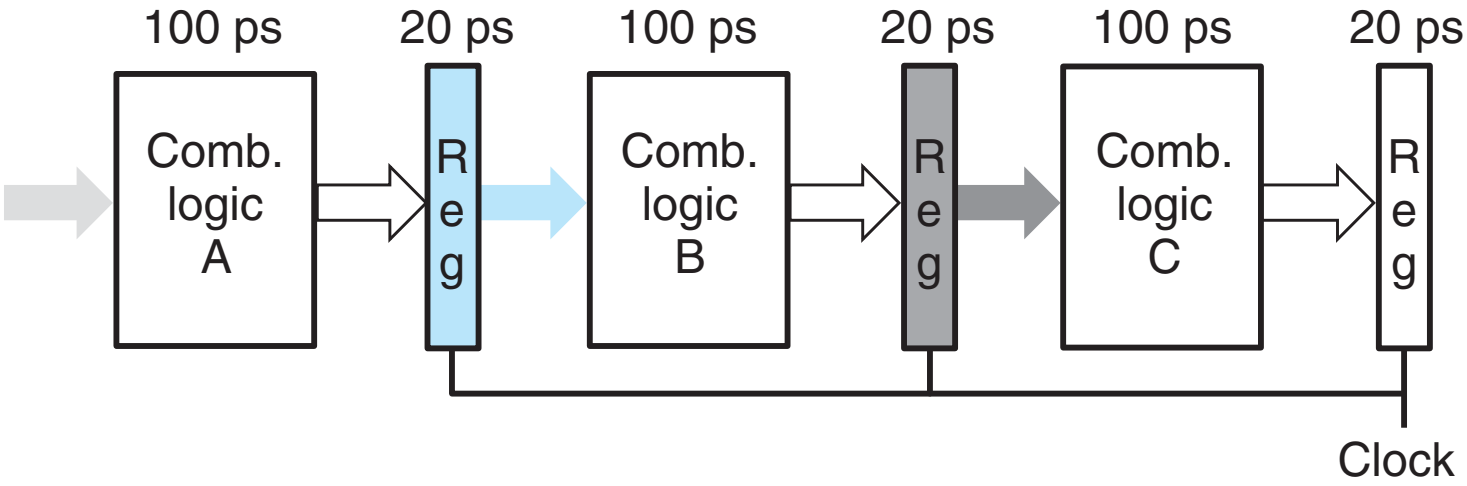
① Time = 239



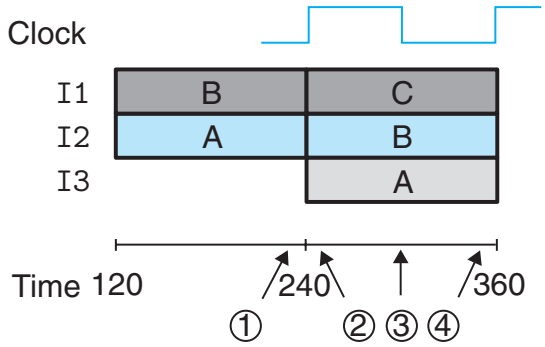
Pipelined Instruction Processing



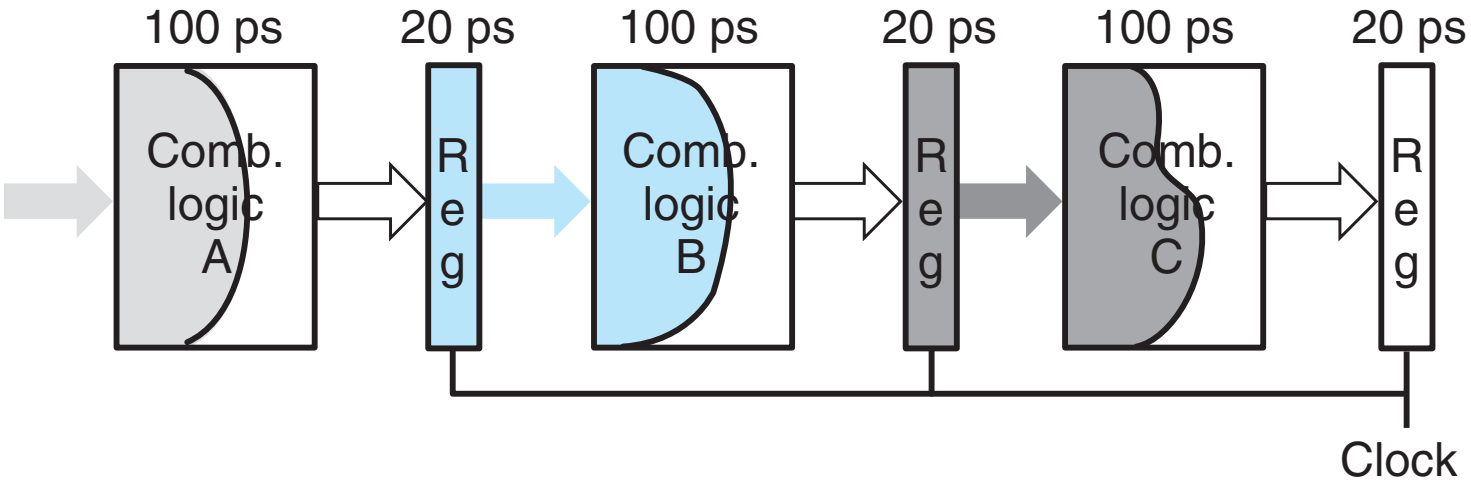
② Time = 241



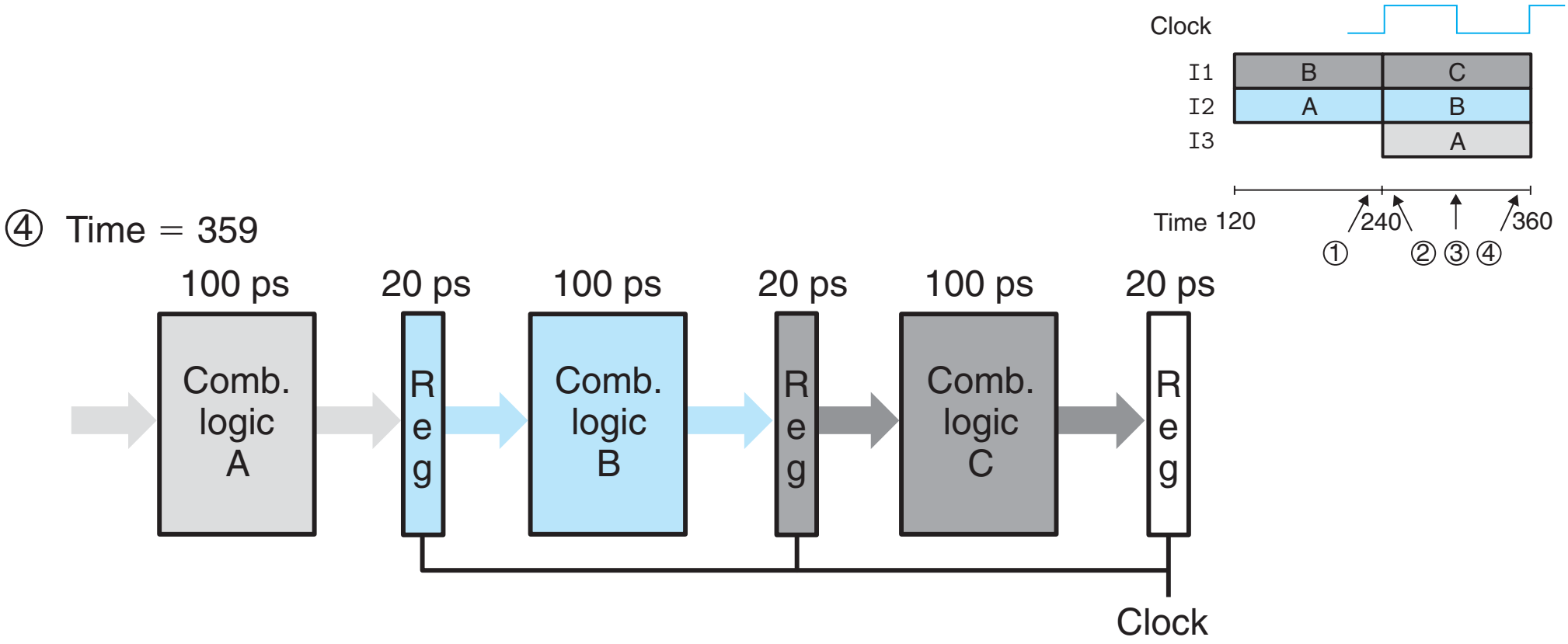
Pipelined Instruction Processing



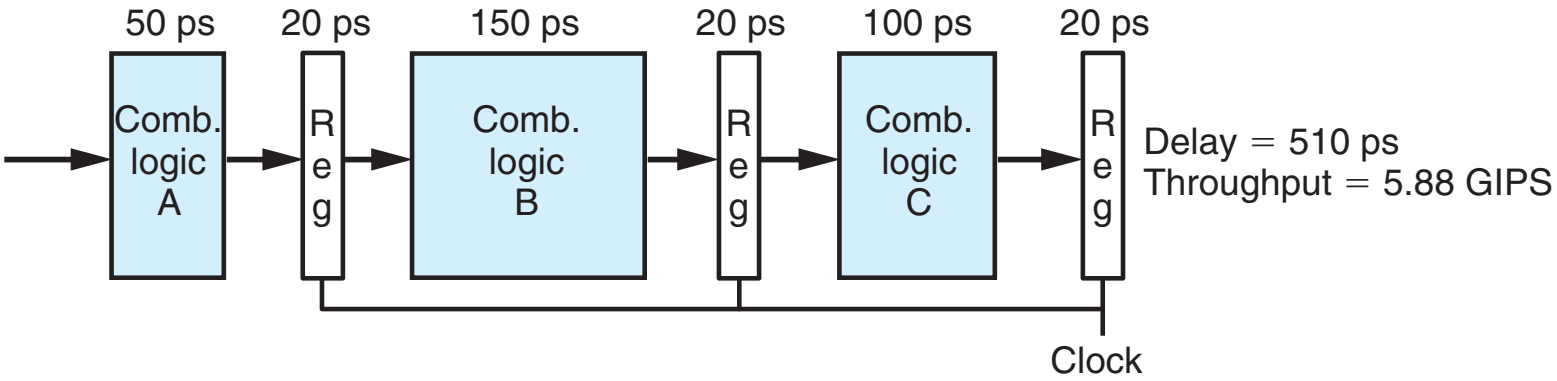
③ Time = 300



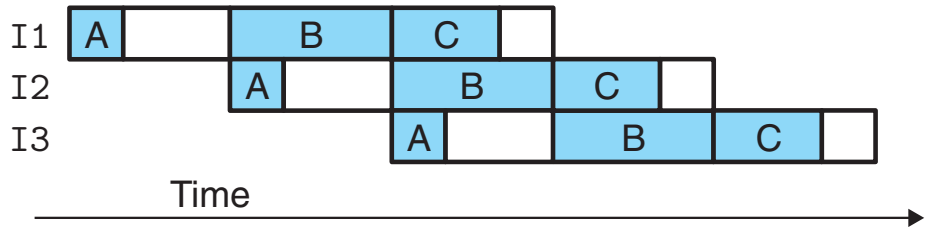
Pipelined Instruction Processing



Non-uniform Partitioning of Stages

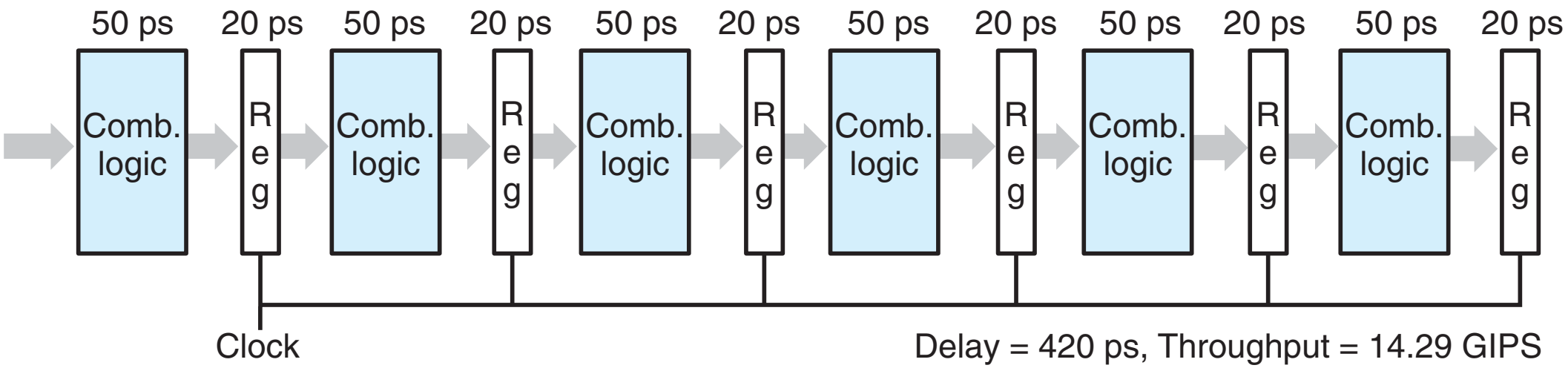


(a) Hardware: Three-stage pipeline, nonuniform stage delays

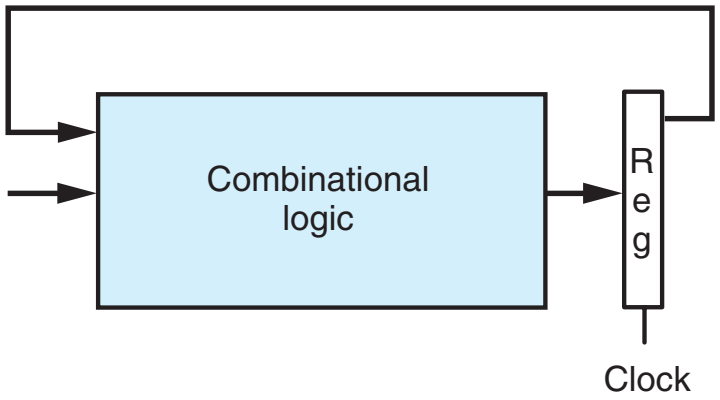


(b) Pipeline diagram

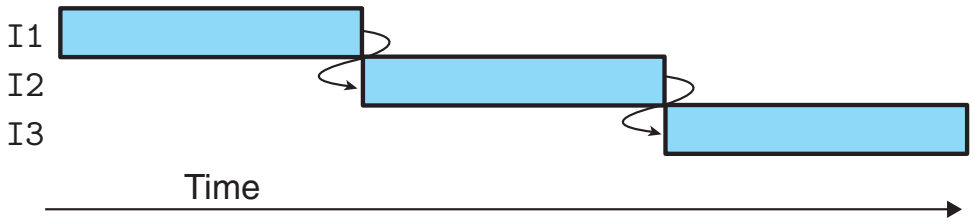
Diminishing returns from Deep Pipeline



Pipeline System with a Feedback



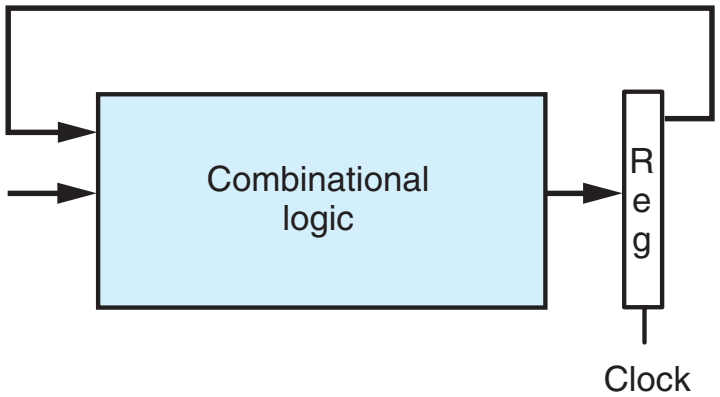
(a) Hardware: Unpipelined with feedback



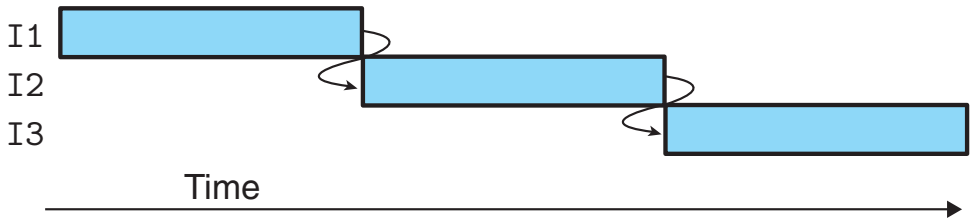
(b) Pipeline diagram

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

Pipeline System with a Feedback



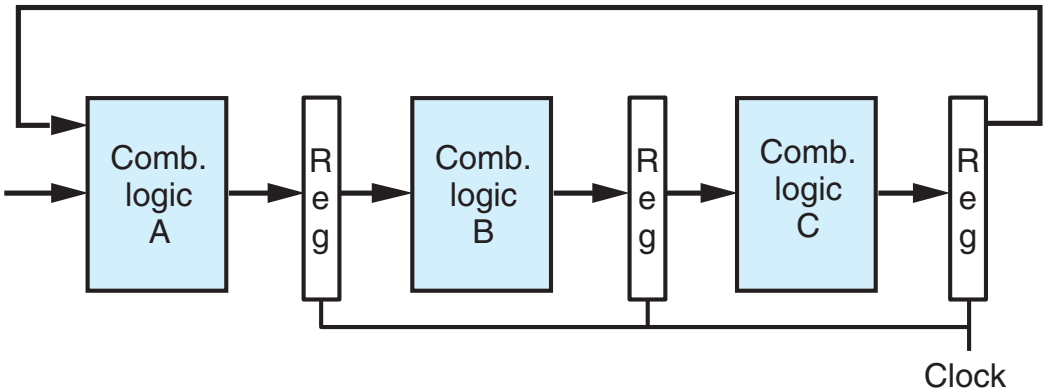
(a) Hardware: Unpipelined with feedback



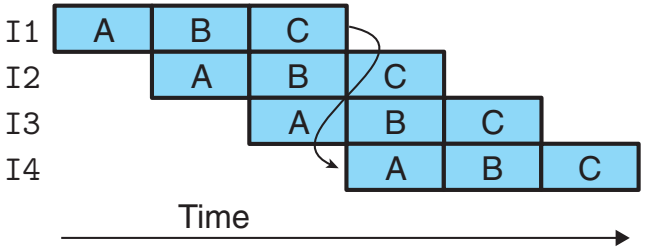
(b) Pipeline diagram

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

Pipeline System with a Feedback



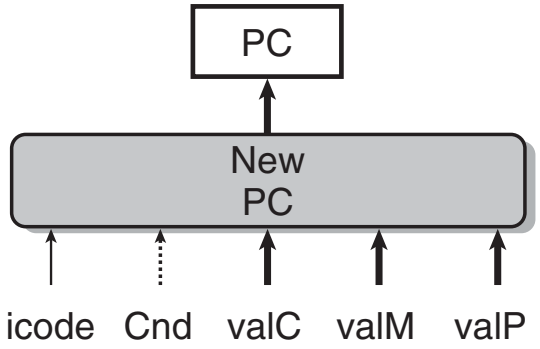
(c) Hardware: Three-stage pipeline with feedback

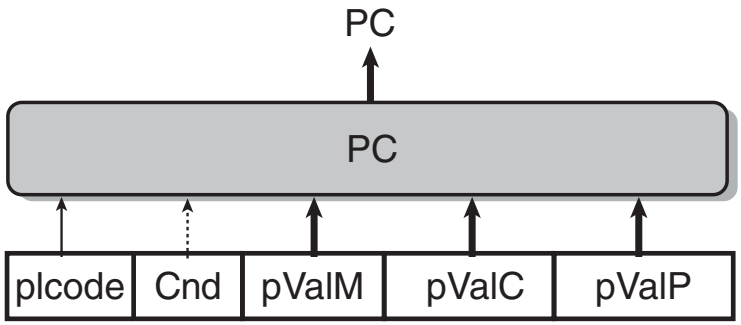


(d) Pipeline diagram

Pipelined Y86: SEQ+

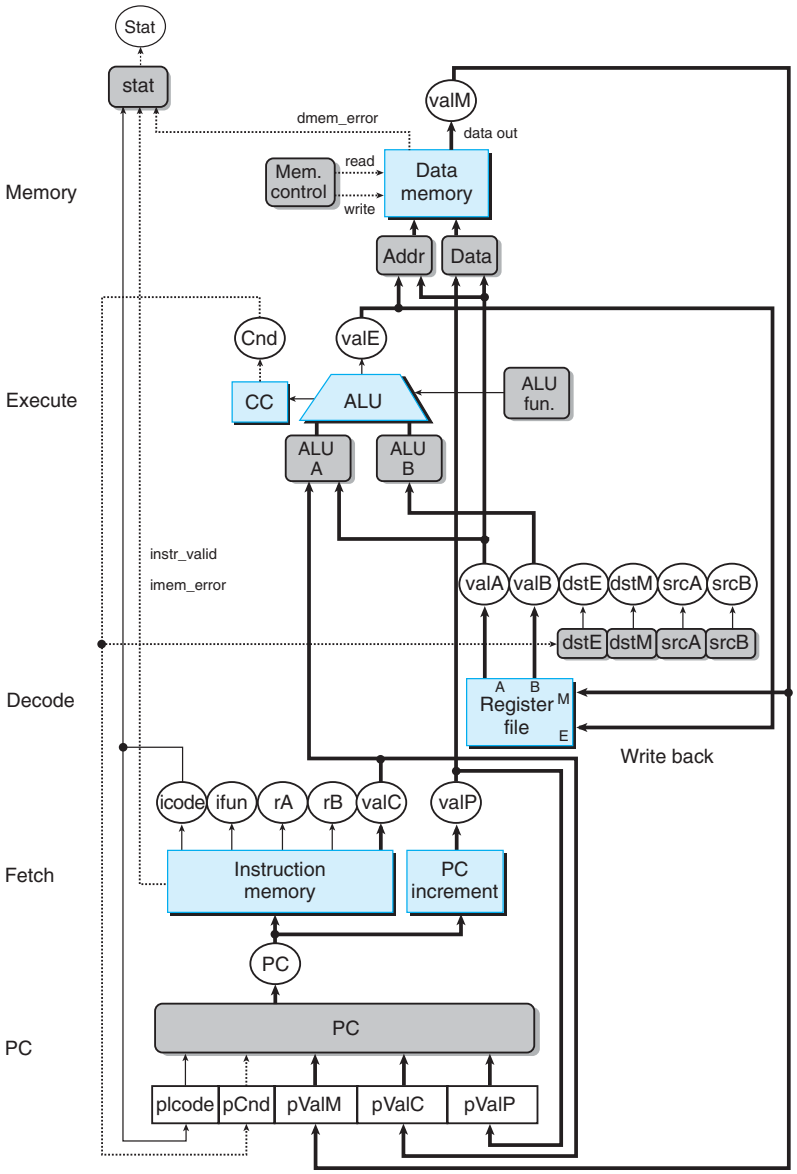
SEQ+ move the PC update stage so that its logic is active at the beginning of the clock cycle by making it compute the PC value for the *current* instruction.





Pipelined Y86: SEQ+

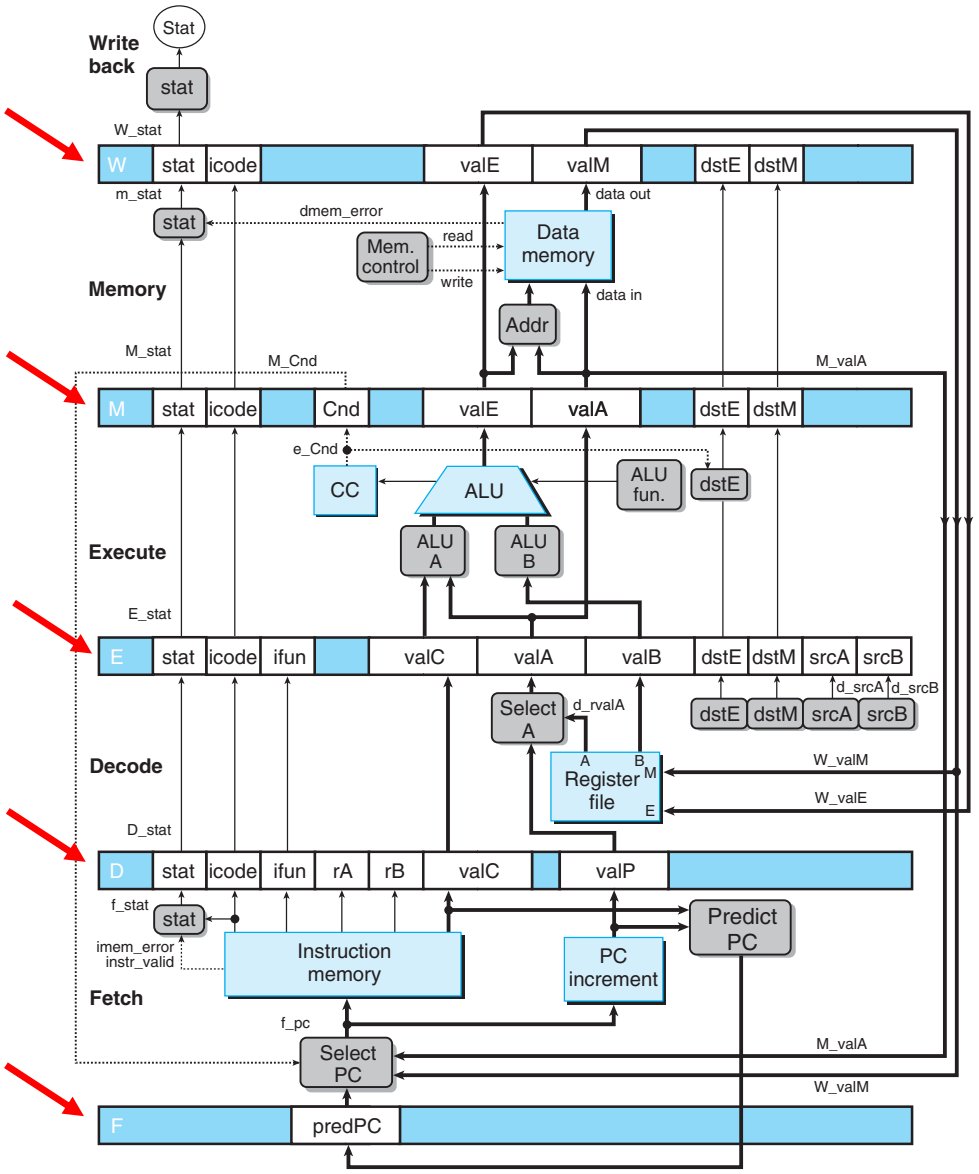
Shifting the PC computation from the end of the clock cycle to the beginning makes it more suitable for pipelining.



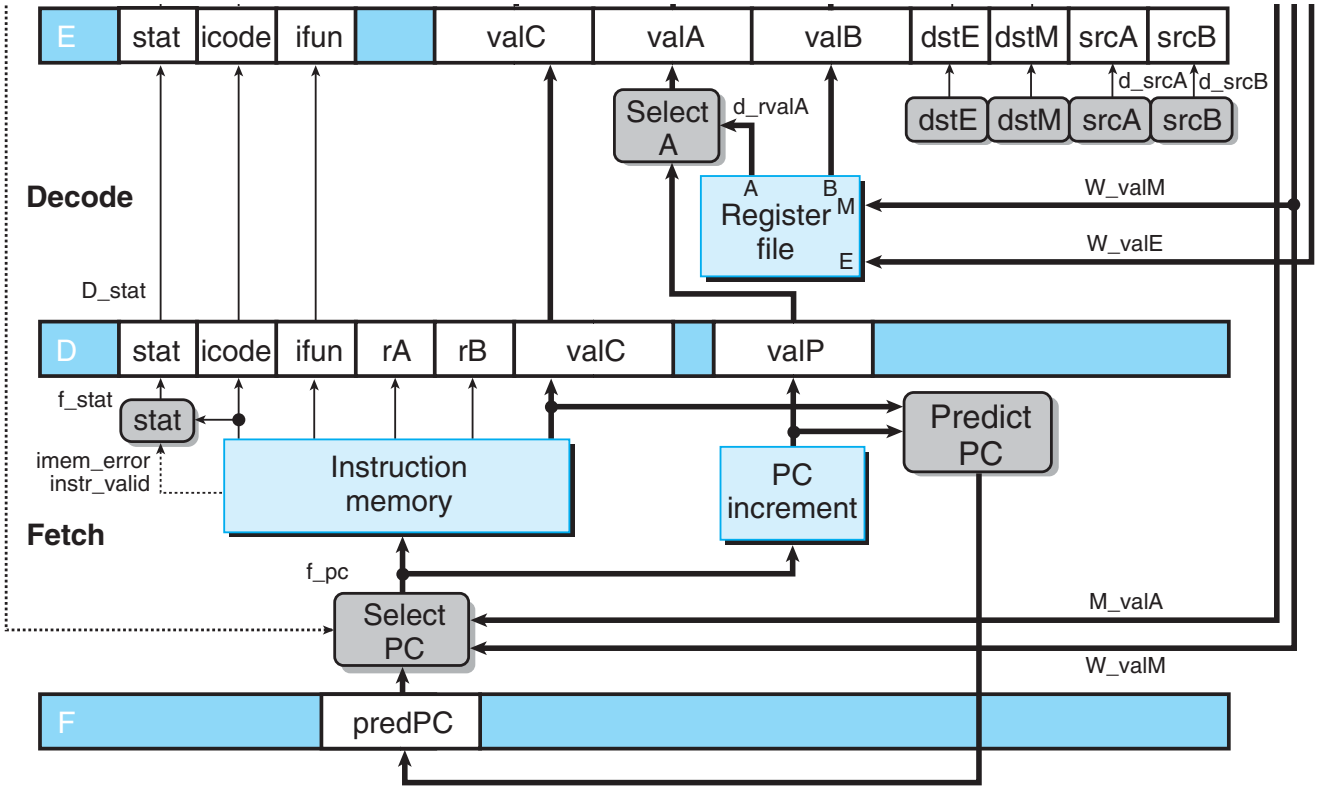
Pipelined Y86: PIPE-

Hardware structure of PIPE–

It is a ***five-stage*** pipeline obtained by inserting pipeline registers into SEQ+ .



Pipelined Y86: PIPE-



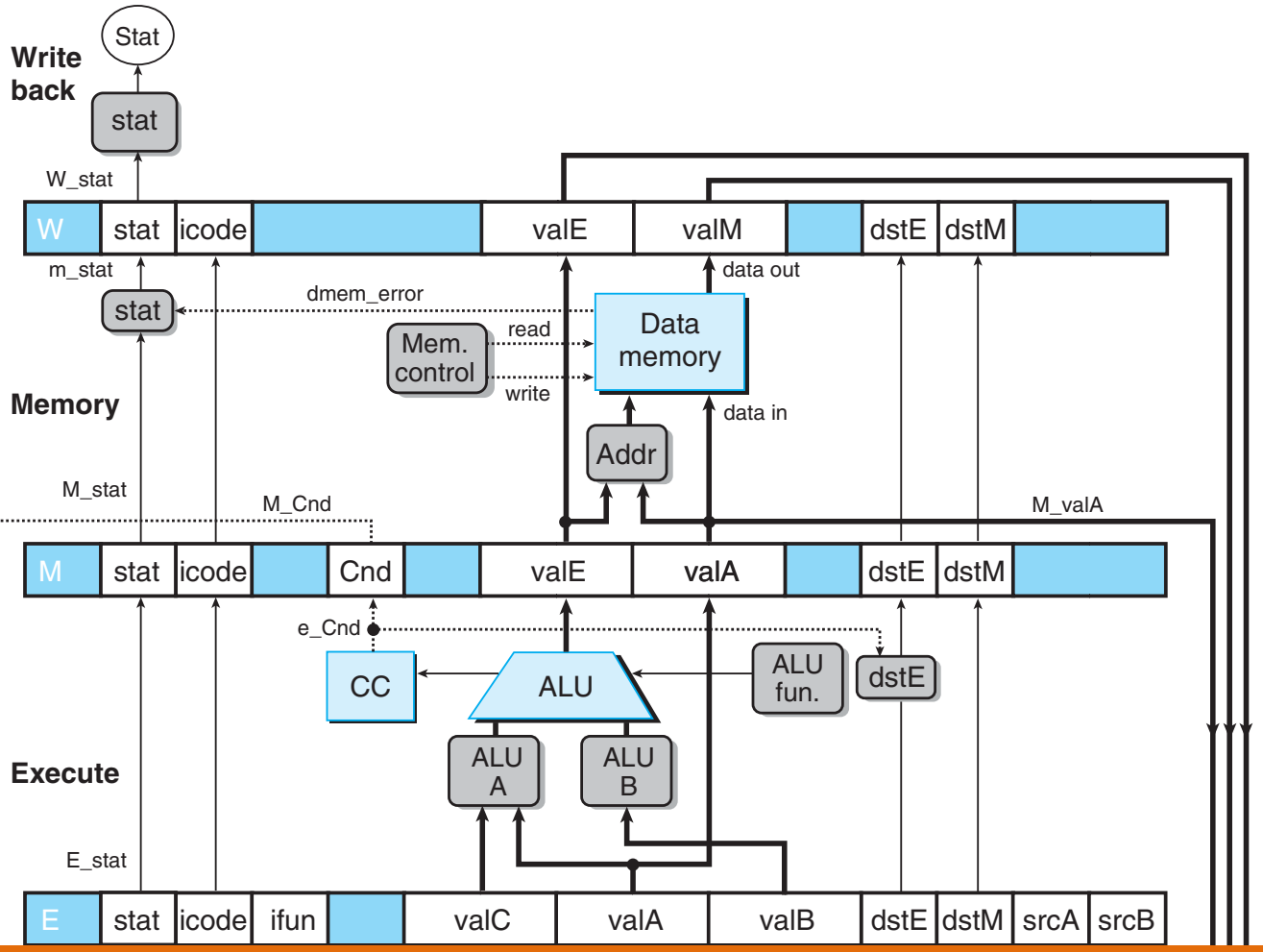
Pipelined Y86: PIPE-

As a general principle, we want to keep all of the information about a particular instruction contained within a single pipeline stage.

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.

Pipelined Y86: PIPE-



Pipelined Y86: PIPE-

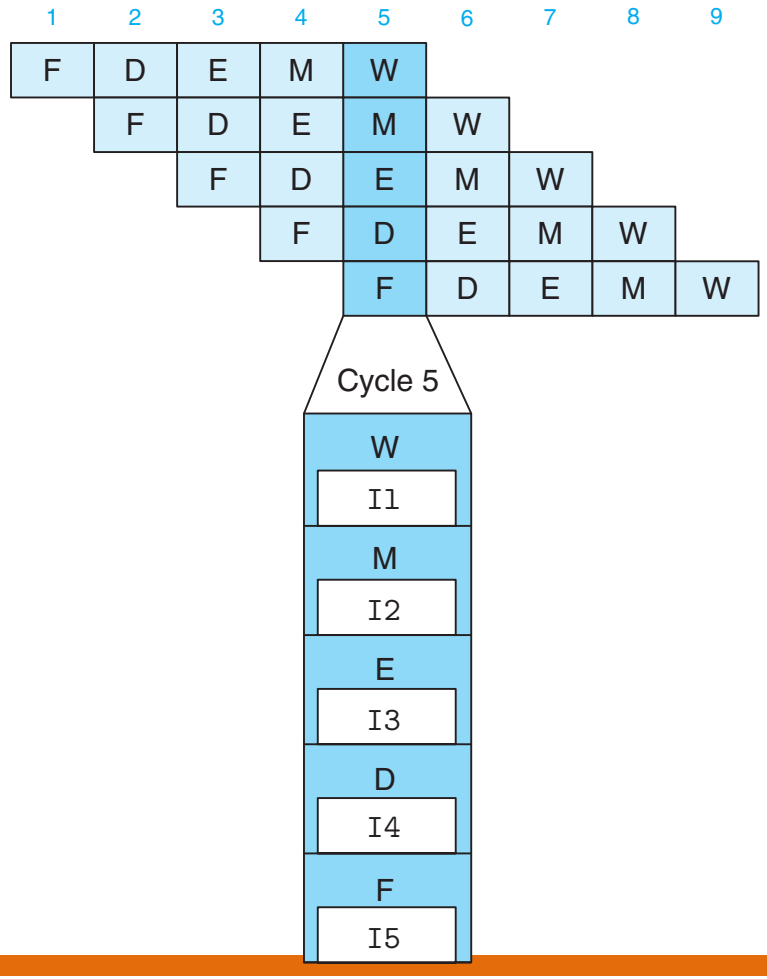
E sits between the decode and execute stages. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage.

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.

Pipelined Y86: PIPE- Instruction Flow

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

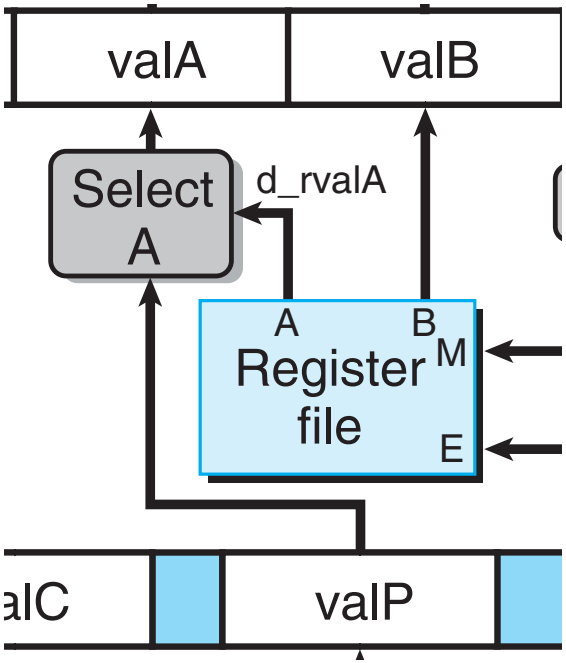


Y86 PIPE-: Rearranging and Relabeling Signals

- A naming scheme is adopted where a signal stored in a pipeline register can be uniquely identified by prefixing its name with that of the pipe register written in uppercase. For example, the four status codes are named **D_stat, E_stat, M_stat,** and **W_stat**.
- Such scheme will avoid a scenario, such as storing the result computed for one instruction at the destination register specified by another instruction.
- 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 lowercase. For example, the output of control logic blocks **stat** in fetch and memory stage is named as **f_stat** and **m_stat**

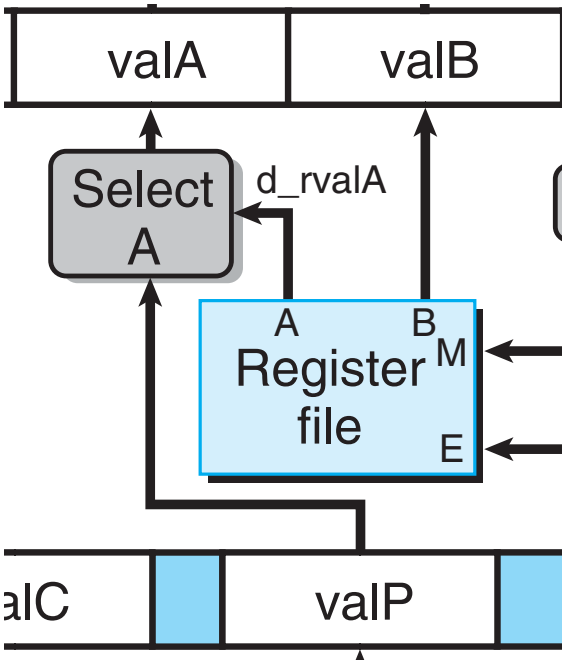
Y86 PIPE-: Rearranging and Relabeling Signals

- 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.



Y86 PIPE-: Rearranging and Relabeling Signals

- Of all the different instructions, only the call requires valP in the memory stage and 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



Y86 PIPE-: Next PC Prediction

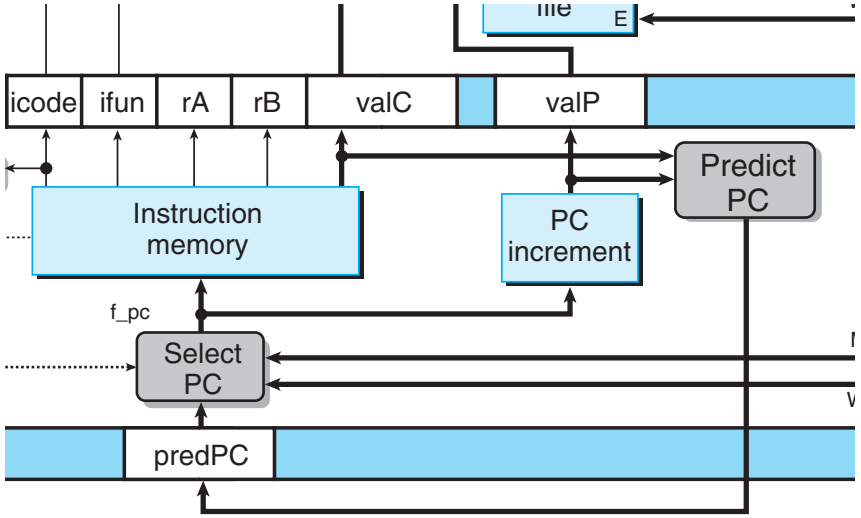
- In pipelined hardware architecture, if the fetched instruction is a conditional branch (eg., **jle**), we will not know whether or not the branch should be taken until several cycles later, after the instruction has passed through the execute stage.
- Similarly, if the fetched instruction is a **ret**, we cannot determine the return location until the instruction has passed through the memory stage.
- Except conditional jump 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.

Y86 PIPE-: Next PC Prediction

- 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 instruction 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.
- This technique of guessing the branch direction and then initiating the fetching of instructions according to our guess is known as ***branch prediction***.

Y86 PIPE-: Next PC Prediction

- “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.
- In SEQ+, we choose 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).

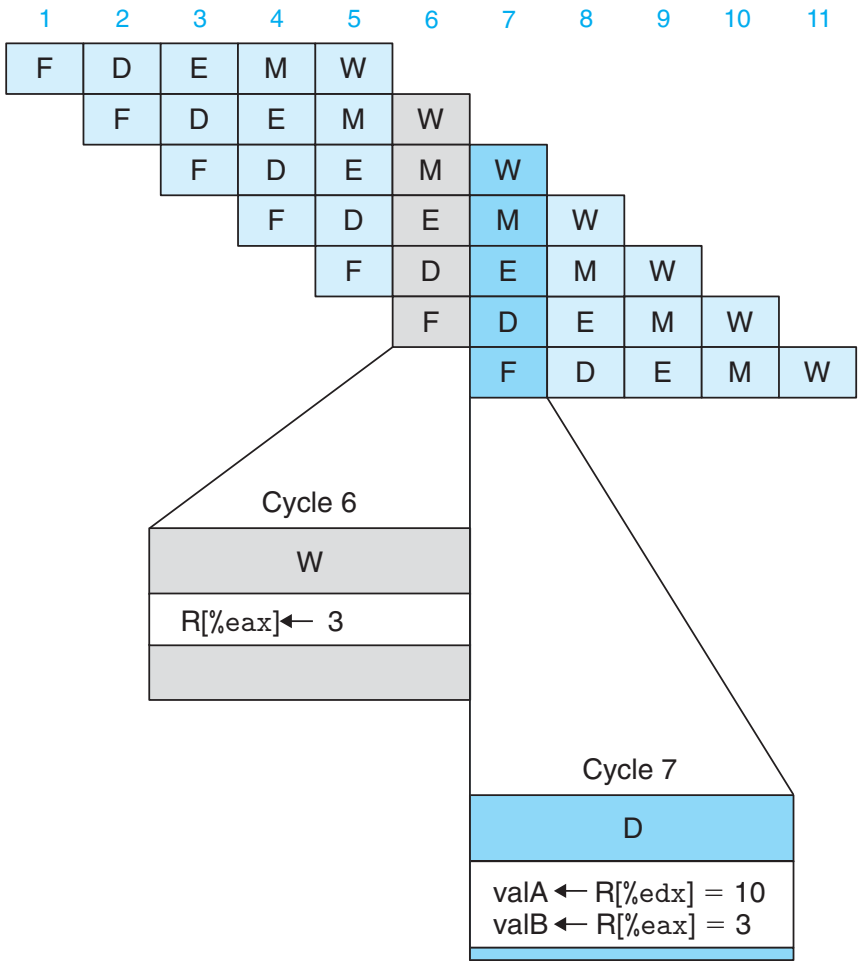


Y86 PIPE-: Pipeline Hazards

- Introducing pipelining into a system with feedback can lead to problems when there are dependencies between successive instructions.
- When such dependencies have the potential to cause an erroneous computation by the pipeline, they are called *hazards.*
- These dependencies can take two forms:
  - *data* dependencies, where the results computed by one instruction are used as the data for a following instruction, also known as ***data hazards***.
  - *control* dependencies, where one instruction determines the location of the following instruction, such as when executing a jump, call, or return. Also known as ***control hazards***.

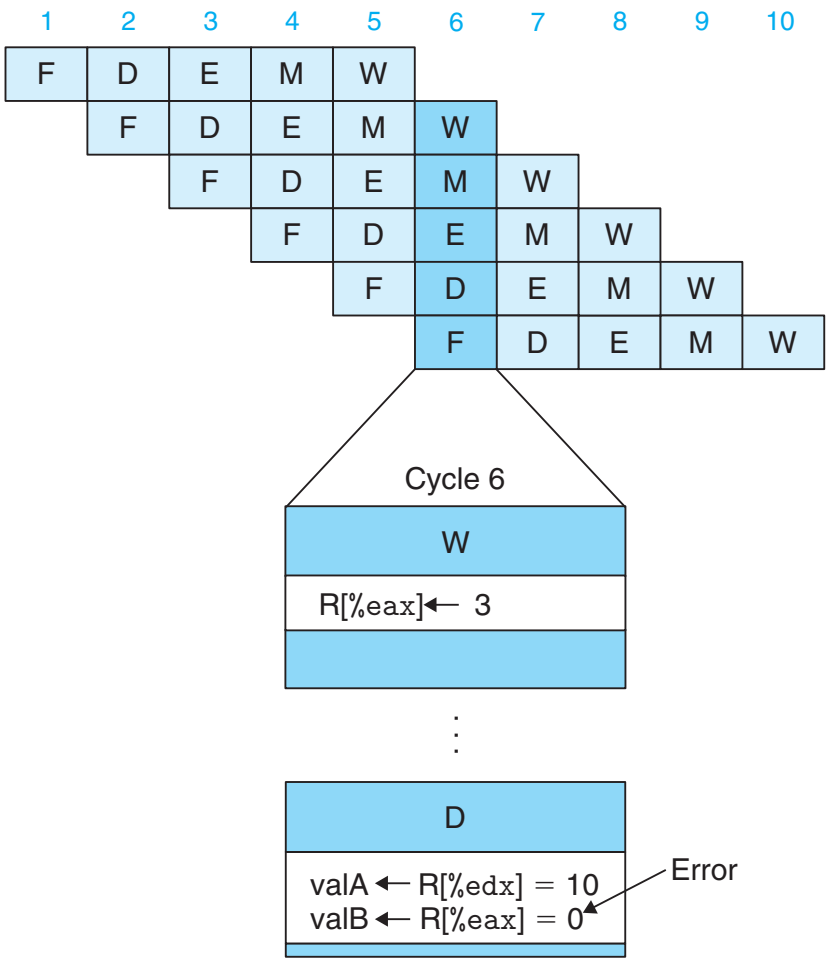
Y86 PIPE-: Data Hazards

```
# progl
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: nop
0x00f: addl %edx,%eax
0x011: halt
```



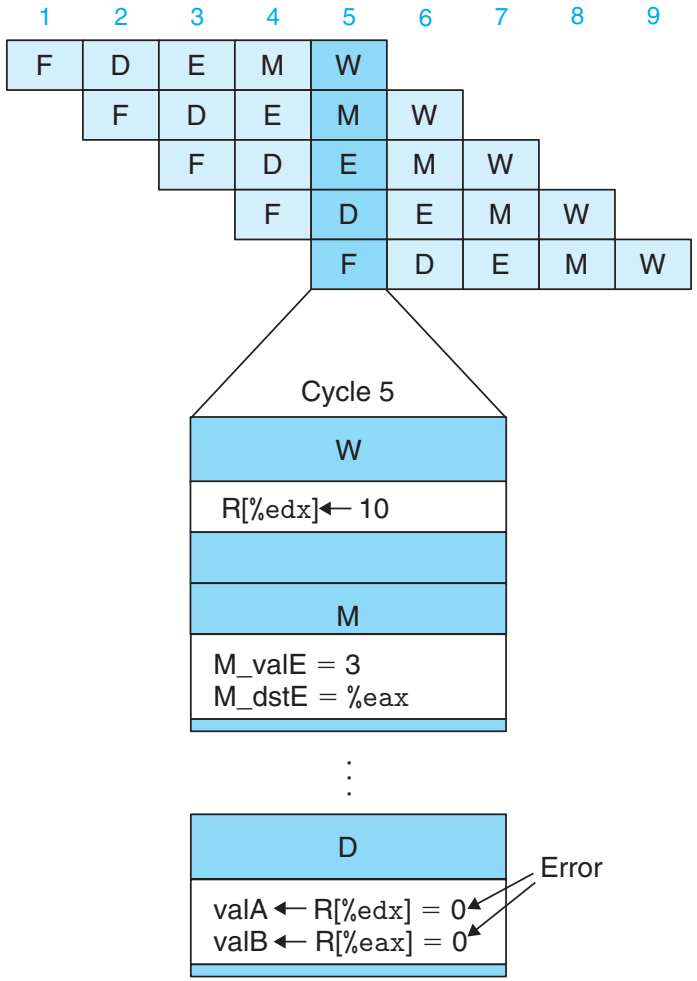
Y86 PIPE-: Data Hazards

```
# prog2
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt
```



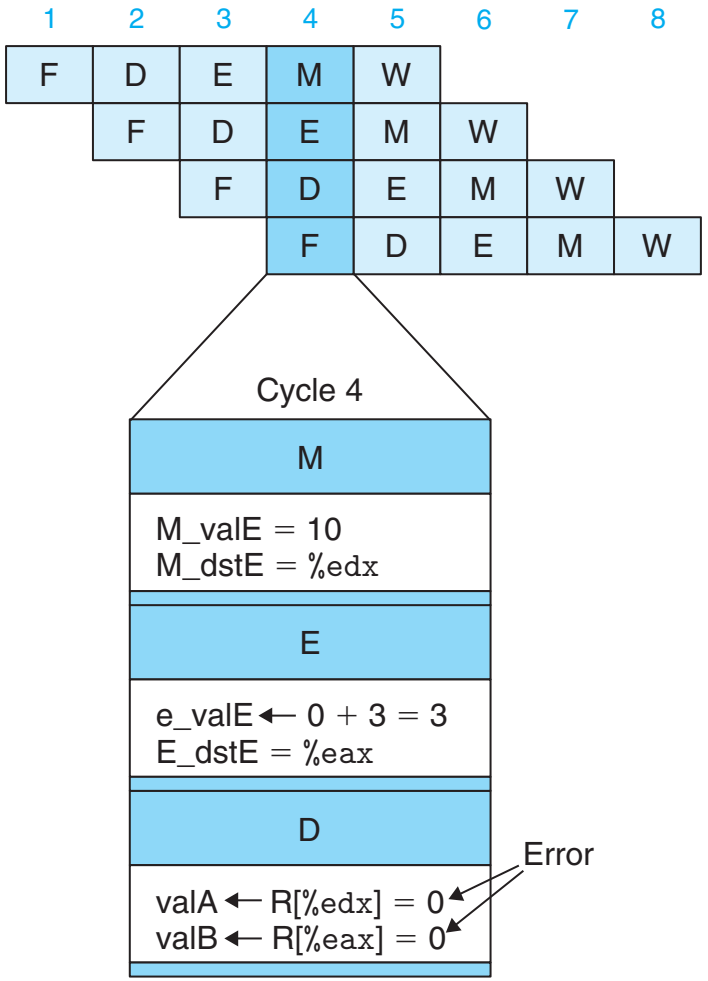
Y86 PIPE-: Data Hazards

```
# prog3
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: addl %edx,%eax
0x00f: halt
```



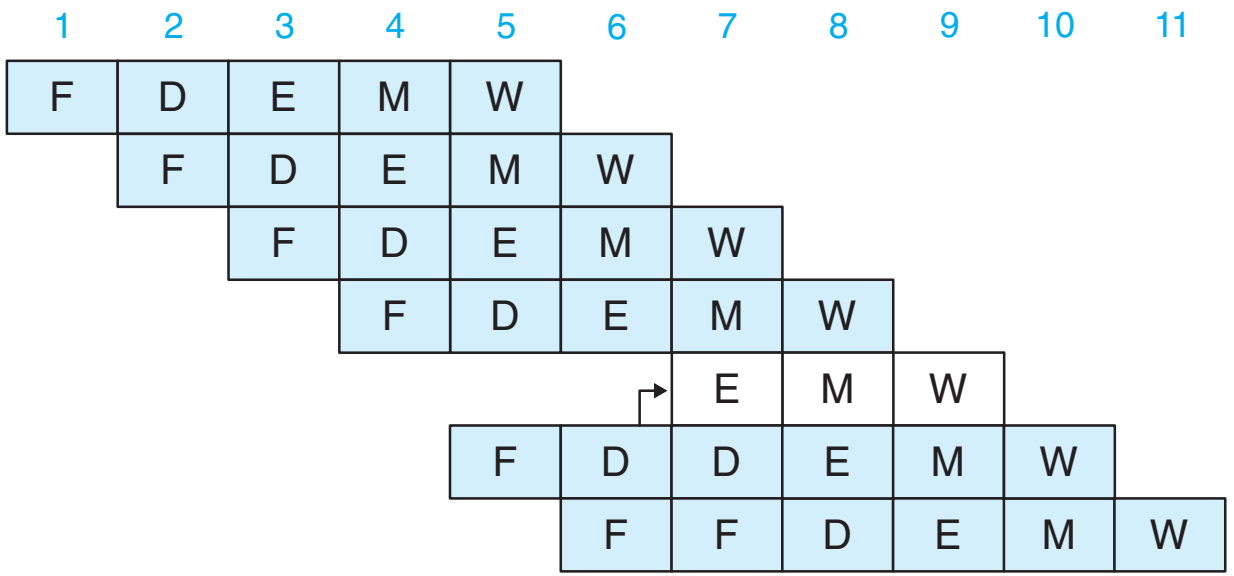
Y86 PIPE-: Data Hazards

```
# prog4
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```



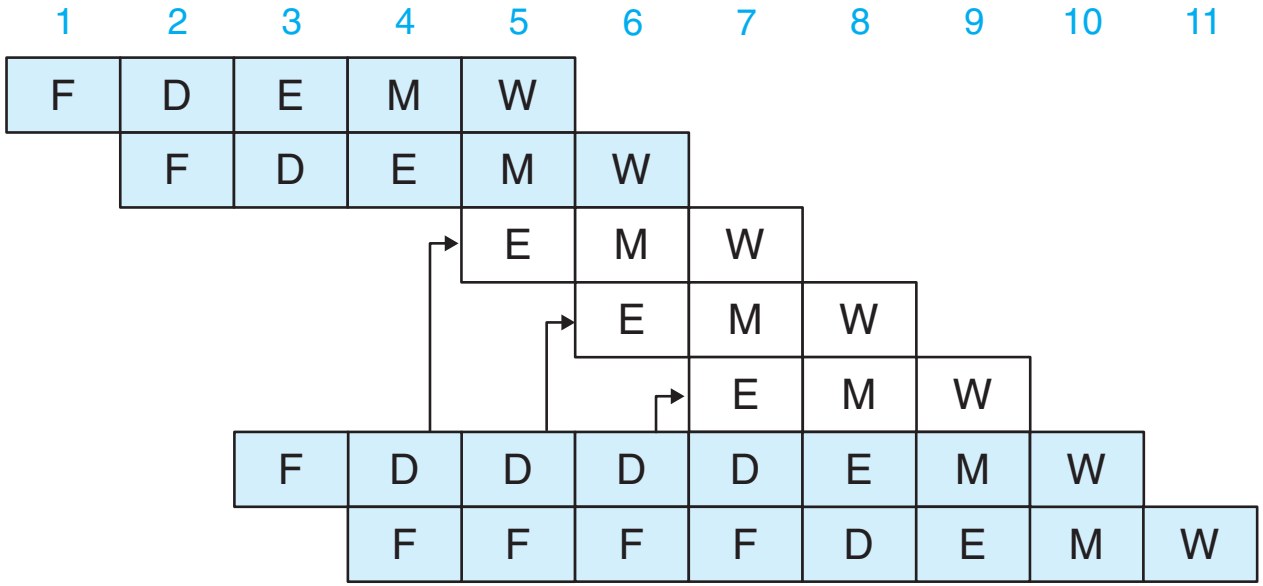
Y86 PIPE-: Avoiding Data Hazards by Stalling

```
# prog2
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
       bubble
0x00e: addl %edx,%eax
0x010: halt
```



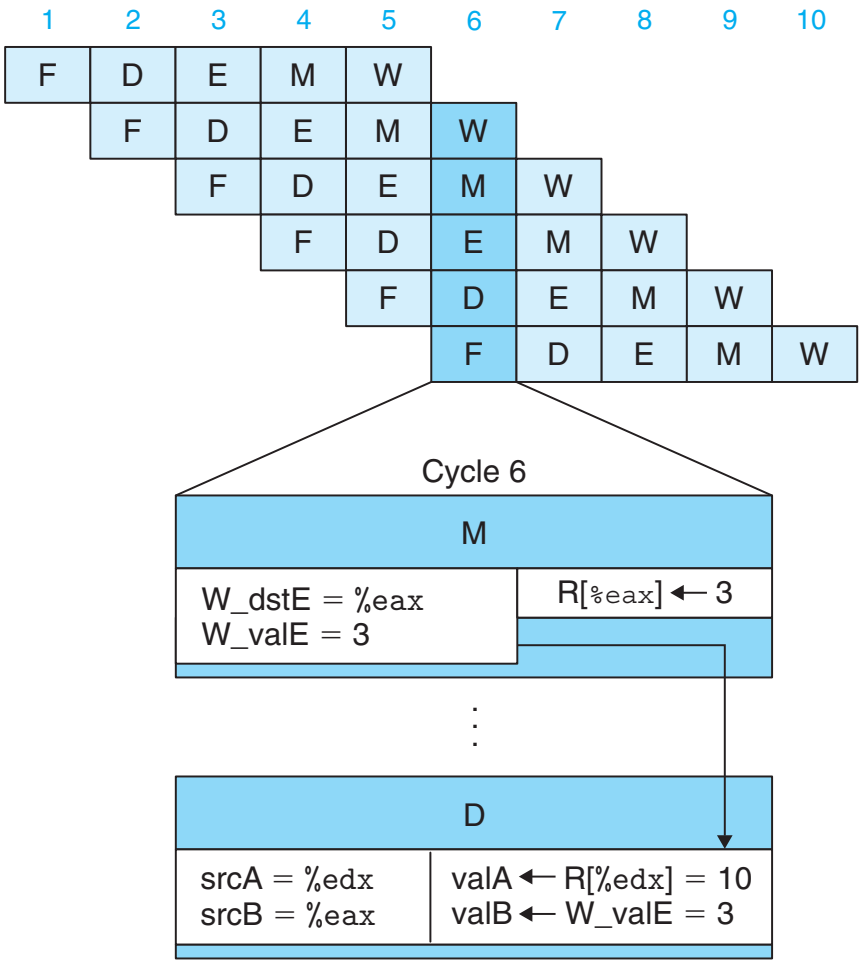
Y86 PIPE-: Avoiding Data Hazards by Stalling

```
# prog4
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
       bubble
       bubble
       bubble
0x00c: addl %edx,%eax
0x00e: halt
```



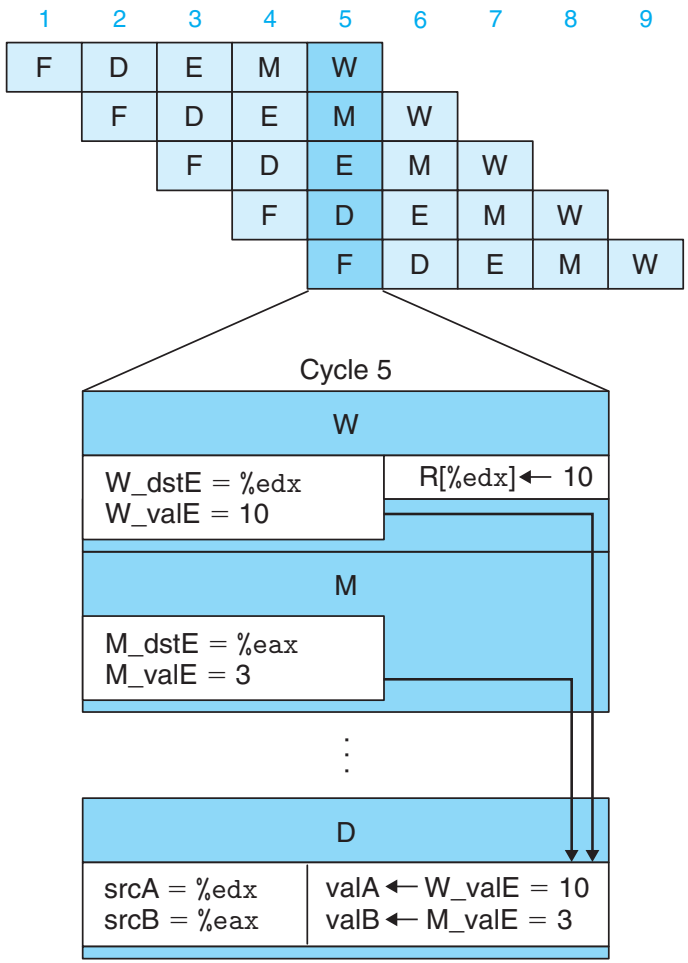
Y86 PIPE-: Avoiding Data Hazards by Forwarding

```
# prog2
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt
```



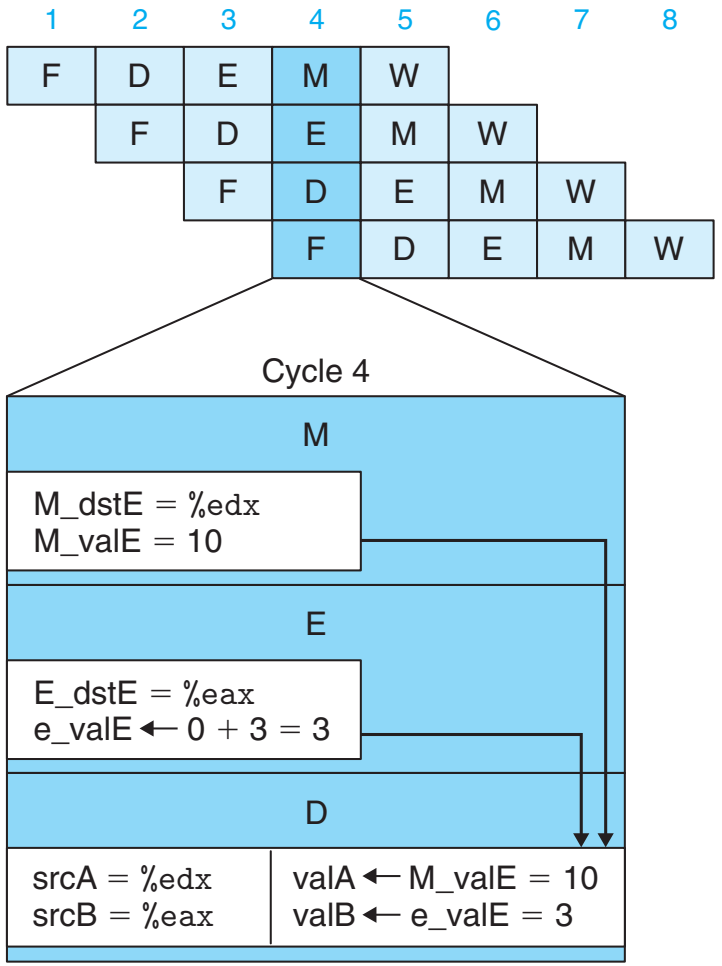
Y86 PIPE-: Avoiding Data Hazards by Forwarding

```
# prog3
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: addl %edx,%eax
0x00f: halt
```



Y86 PIPE-: Avoiding Data Hazards by Forwarding

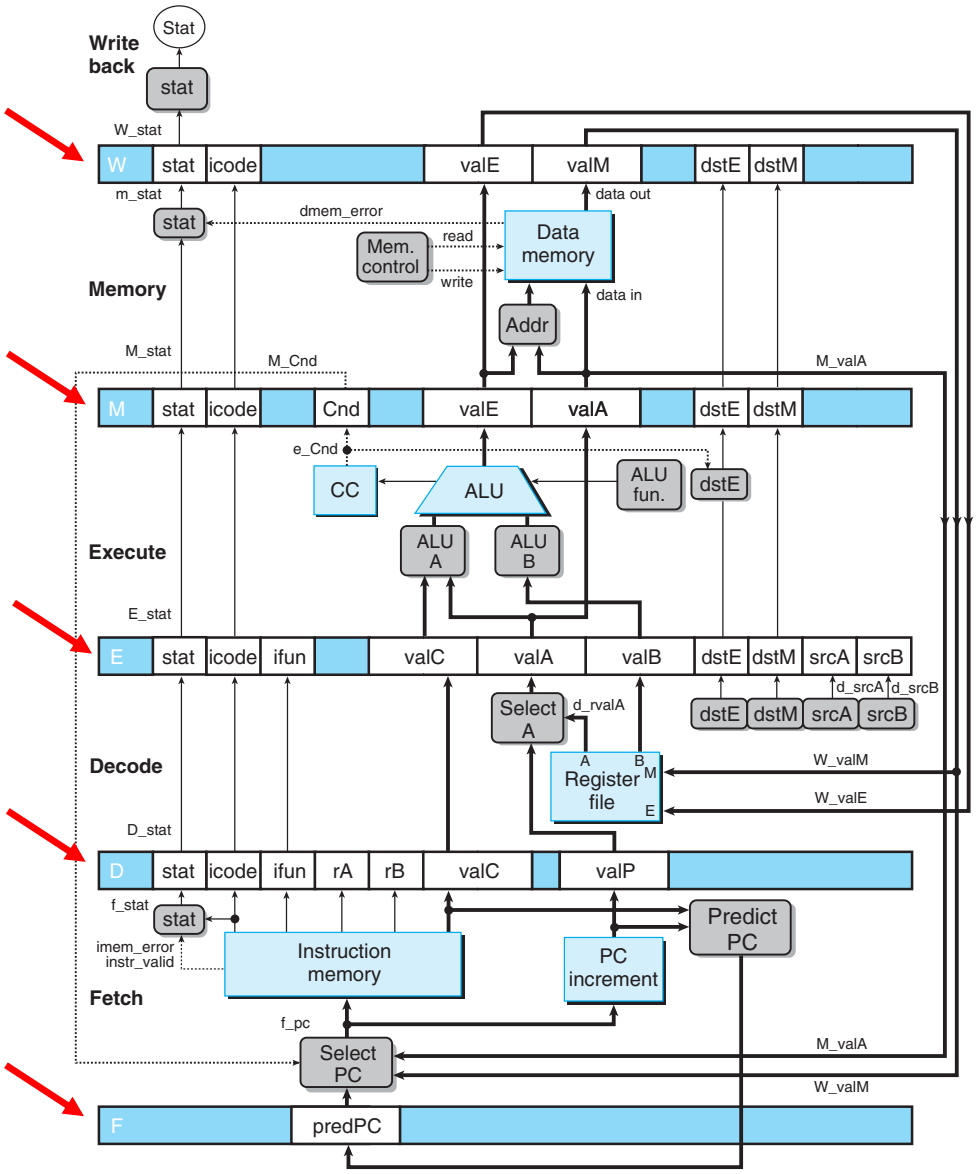
```
# prog4
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```



Pipelined Y86: PIPE-

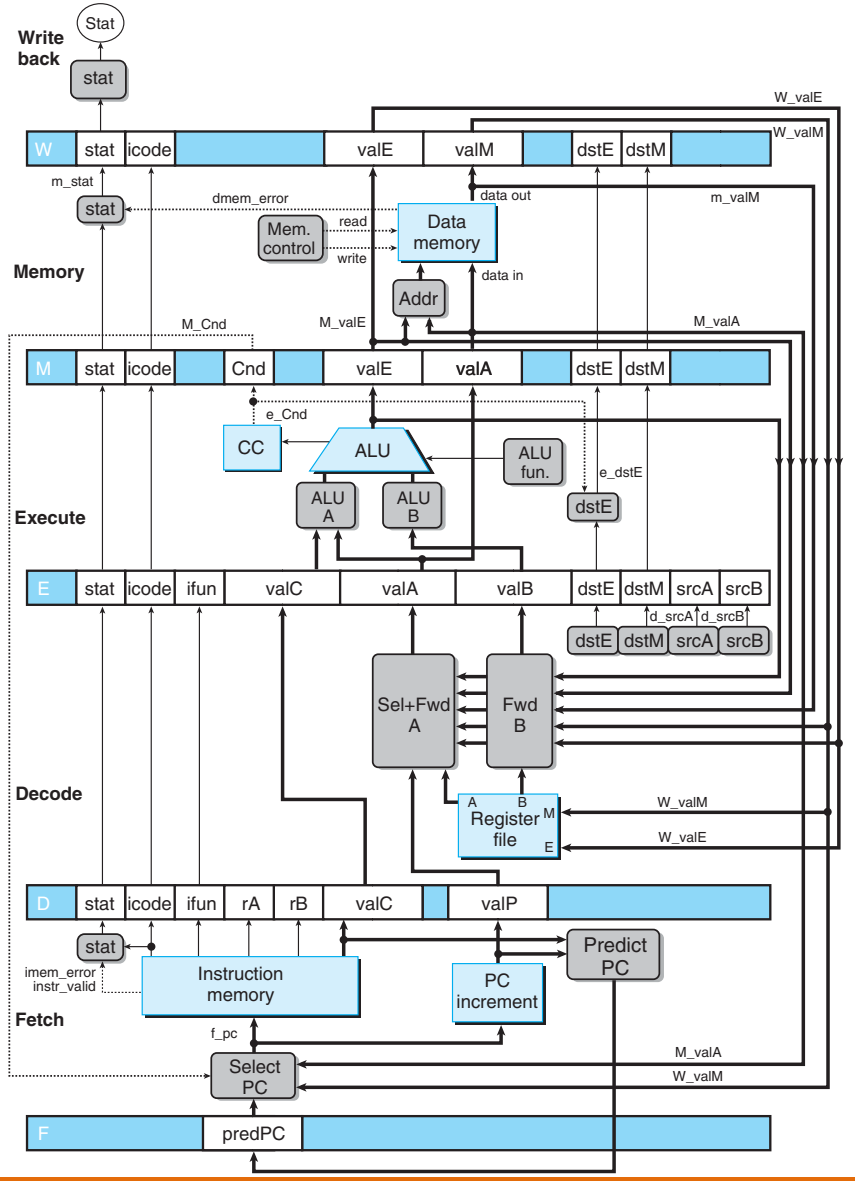
Hardware structure of PIPE–

It is a ***five-stage*** pipeline obtained by inserting pipeline registers into SEQ+ .



Y86 PIPE

- Unlike PIPE-, the new PIPE hardware architecture can handle data hazards with forwarding.



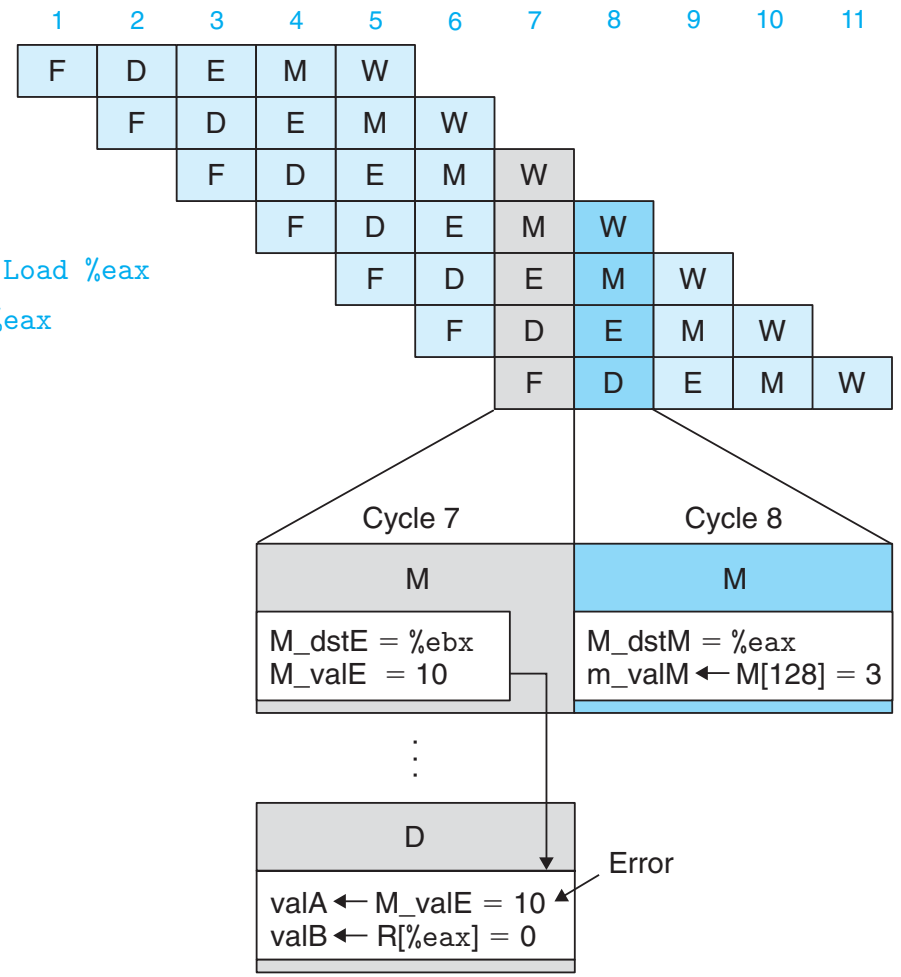
Y86 PIPE: Load/Use Data Hazards

- One class of data hazards cannot be handled purely by forwarding, because memory reads occur late in the pipeline.
- We can avoid a load/use data hazard with a combination of stalling and forwarding.

```
# prog5
0x000: irmovl $128,%edx
0x006: irmovl  $3,%ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10,%ebx
0x018: mrmovl 0(%edx),%eax # Load %eax
0x01e: addl %ebx,%eax # Use %eax
0x020: halt
```

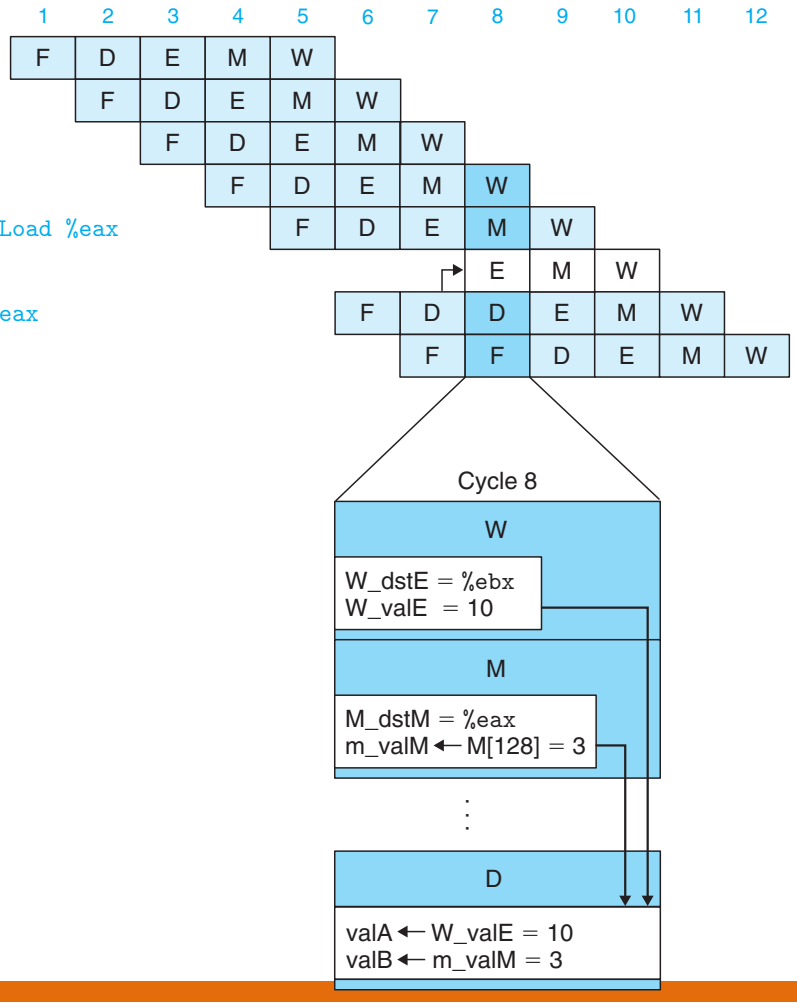
Y86 PIPE-: Load/Use Data Hazards

```
# prog5
0x000: irmovl $128,%edx
0x006: irmovl  $3,%ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10,%ebx
0x018: mrmovl 0(%edx),%eax # Load %eax
0x01e: addl %ebx,%eax # Use %eax
0x020: halt
```



Y86 PIPE-: Load/Use Data Hazards

```
# prog5
0x000: irmovl $128,%edx
0x006: irmovl  $3,%ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10,%ebx
0x018: mrmovl 0(%edx),%eax # Load %eax
     bubble
0x01e: addl %ebx,%eax # Use %eax
0x020: halt
```



Y86 PIPE: Exception Handling

- Exceptions can be generated either internally, by the executing program, or externally, by some outside signal.

- Our instruction set architecture includes three different internally generated exceptions, caused by
  - a halt instruction,
  - an instruction with an invalid combination of instruction and function code,
  - an attempt to access an invalid address, either for instruction fetch or data read or write.
- A more complete processor design would also handle external exceptions.

Y86 PIPE: Exception Handling

- Let us refer to the instruction causing the exception as the ***excepting*** instruction.
- In the case of an invalid instruction address, there is no actual excepting instruction, but it is useful to think of there being a sort of “virtual instruction” at the invalid address.
- In our simplified ISA model, we want the processor to halt when it reaches an exception and to set the appropriate status code.
- It should appear that all instructions up to the excepting instruction have completed, but none of the following instructions should have any effect on the programmer-visible state.

Y86 PIPE: Exception Handling

Case of Multiple Exceptions

- It is possible to have exceptions triggered by multiple instructions simultaneously.
- The basic rule is to put priority on the exception triggered by the instruction that is furthest along the pipeline.
- In terms of the machine-language program, the instruction in the memory stage should appear to execute before one in the fetch stage, and therefore only this exception should be reported to the operating system.

Y86 PIPE: Exception Handling

Case of Exception in Mis-Predicted Branch

- An instruction is first fetched and begins execution, causes an exception, and later is canceled due to a mis-predicted branch.

```
0x000: 6300          |    xorl %eax,%eax
0x002: 740e000000    |    jne  Target       # Not taken
0x007: 30f001000000 |    irmovl $1, %eax   # Fall through
0x00d: 00            |    halt
0x00e:               | Target:
0x00e: ff            |    .byte 0xFF        # Invalid instruction code
```

- The pipeline control logic will cancel such instruction, but we want to avoid raising an exception.

Y86 PIPE: Exception Handling

Case of Exception in Memory or Write-back Stage

- 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.

```
irmovl $1,%eax
xorl %esp,%esp        # Set stack pointer to 0 and CC to 100
pushl %eax            # Attempt to write to 0xfffffffc
addl  %eax,%eax       # (Should not be executed) Would set CC to 000
```

- To avoid having any updating of the programmer-visible state by instructions beyond the excepting instruction, the pipeline control logic must 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.

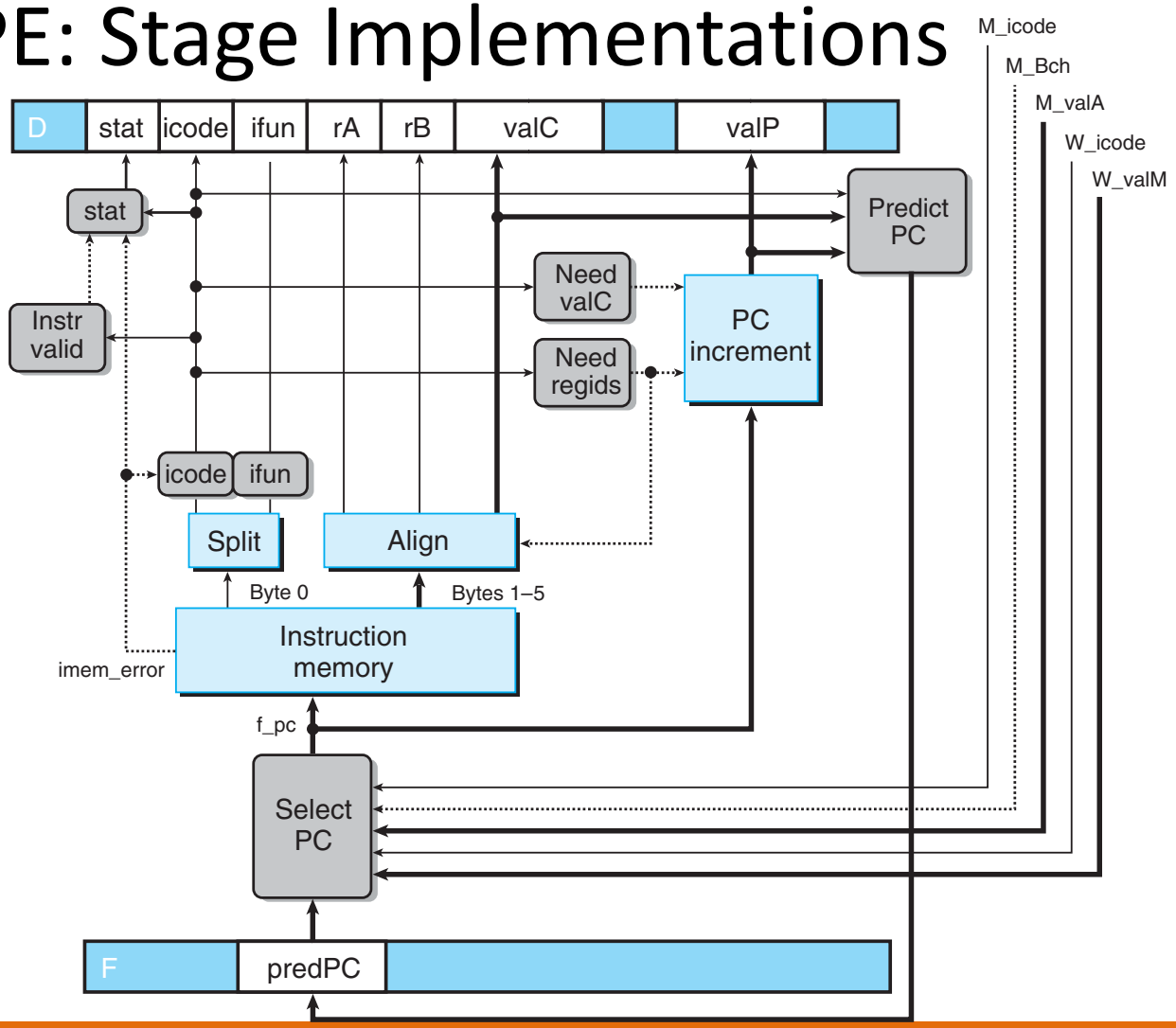
Y86 PIPE: Exception Handling

Handling these 3 types of Exceptions in pipeline architecture

- 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.

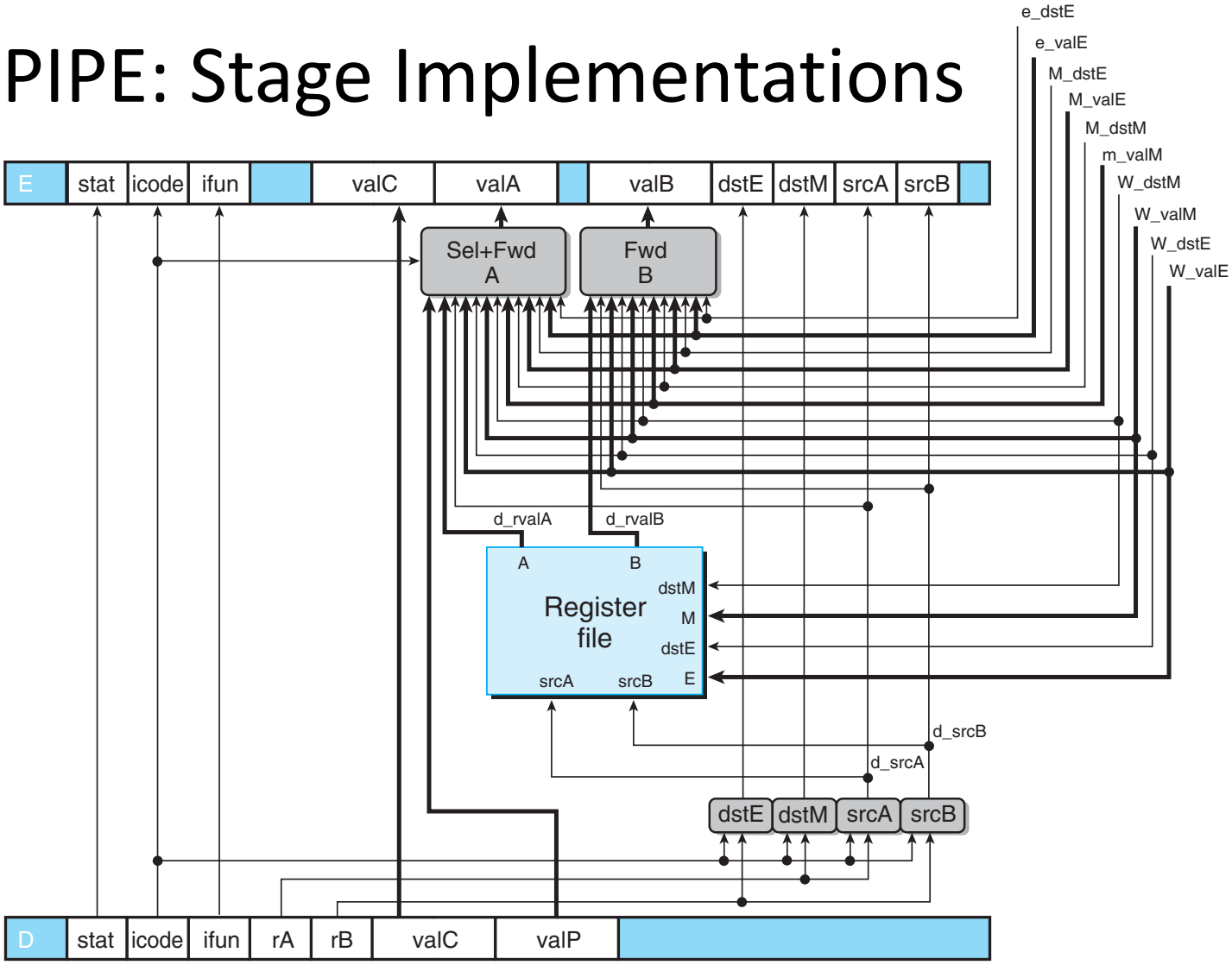
Y86 PIPE: Stage Implementations

PC Selection and Fetch Stage



Y86 PIPE: Stage Implementations

Decode and Write-back Stage



Y86 PIPE: Stage Implementations

Decode and Write-back Stage: Forwarding Priority

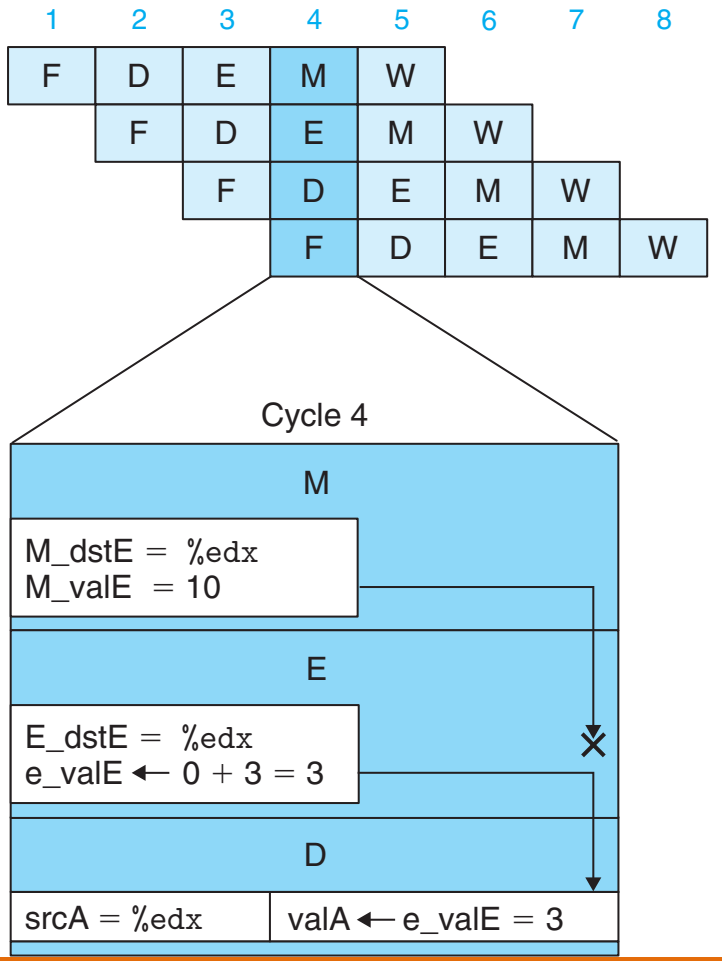
When signal icode matches the instruction code for either call or jXX, this block should select valP as its output. Else, following forwarding options.

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

Y86 PIPE: Stage Implementations

Decode and Write-back Stage: Forwarding Priority

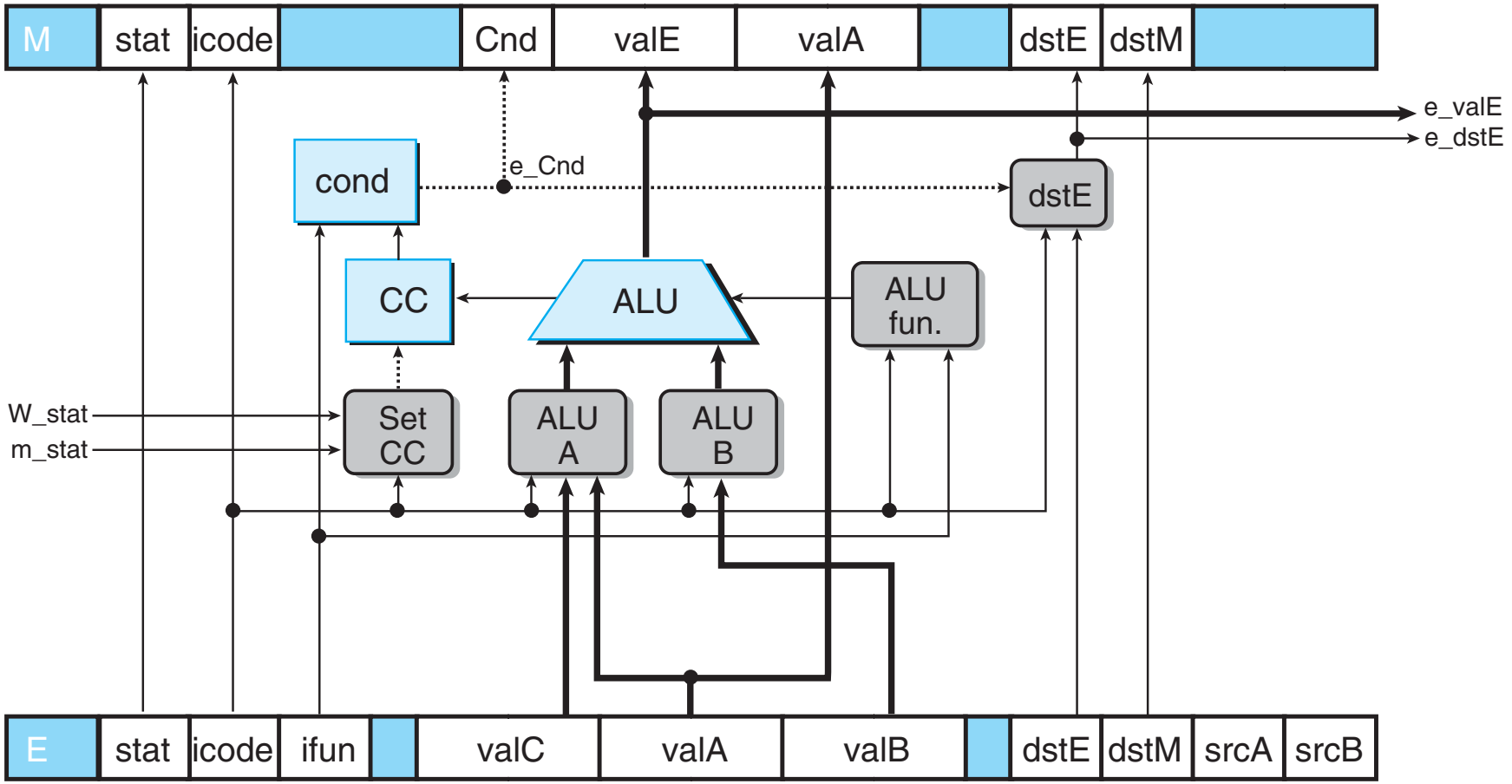
```
# prog6
0x000: irmovl $10,%edx
0x006: irmovl  $3,%edx
0x00c: rrmovl %edx,%eax
0x00e: halt
```



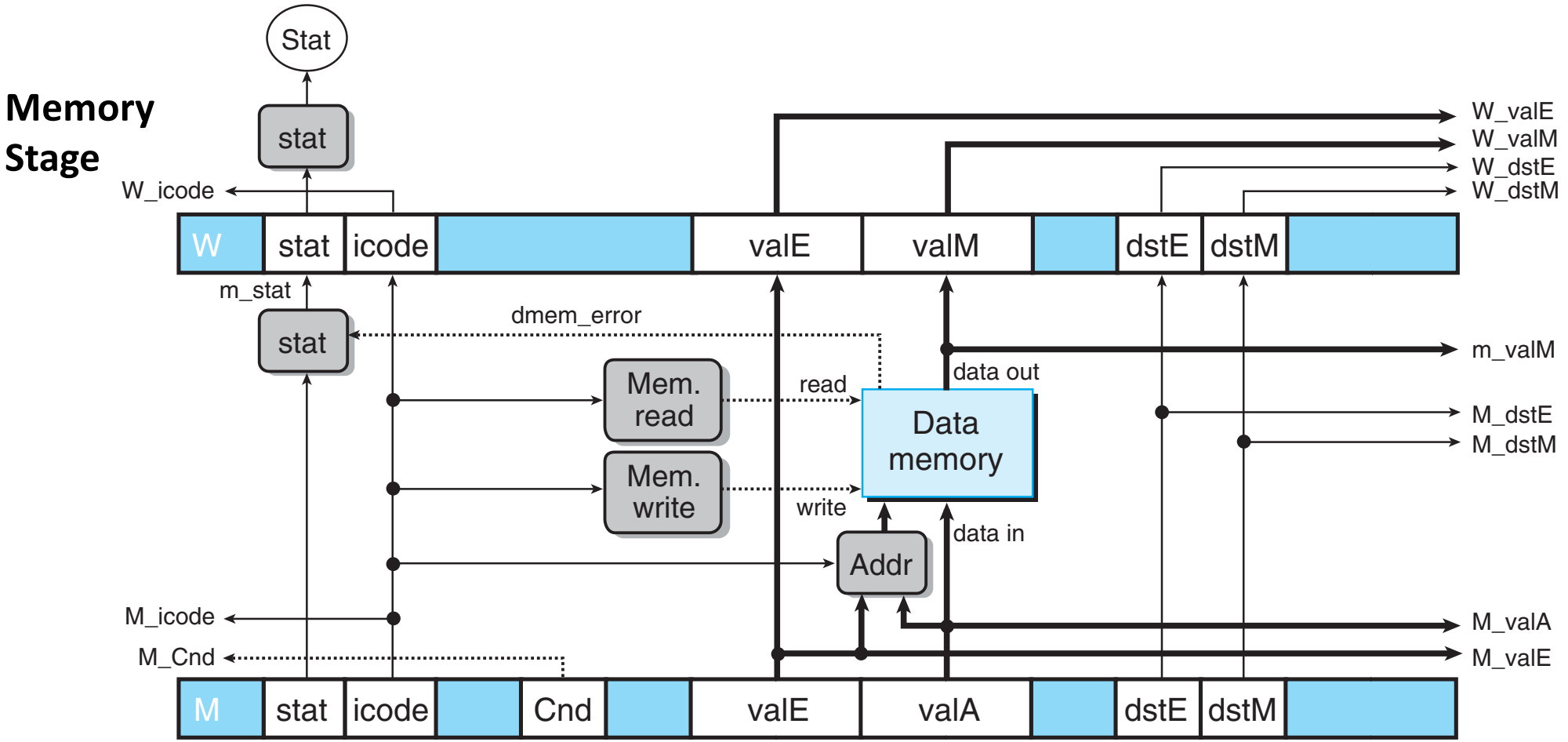
A pipelined implementation should always give priority to the forwarding source in the earliest pipeline stage, since it holds the latest instruction in the program sequence setting the register.

Y86 PIPE: Stage Implementations

Execute Stage



Y86 PIPE: Stage Implementations



Y86 PIPE: Pipeline Control Logic

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.

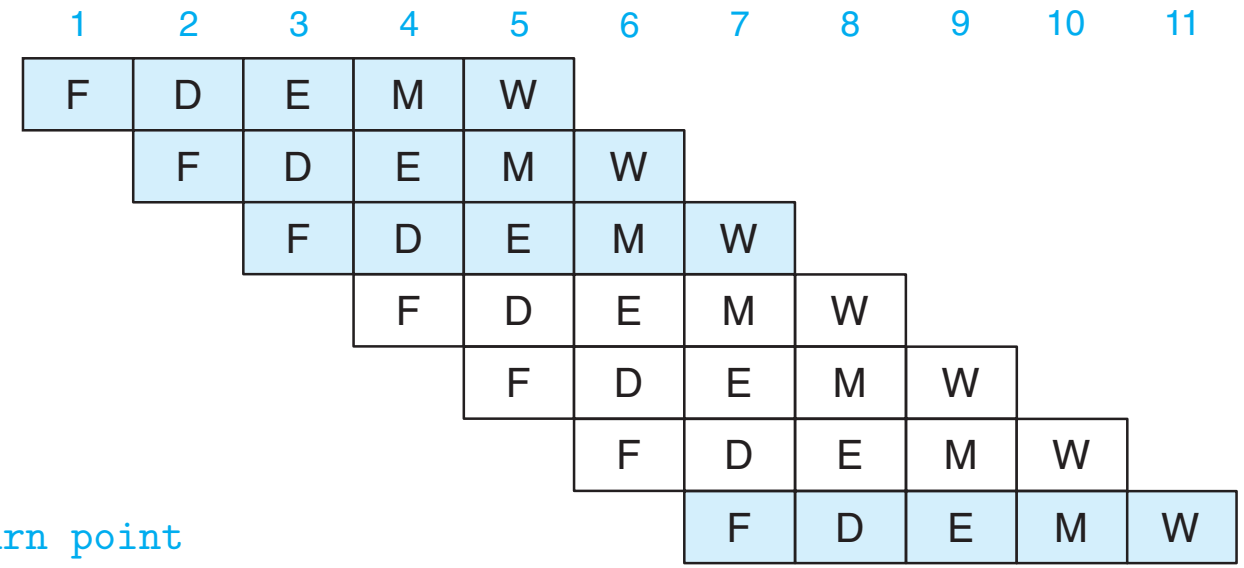
Exceptions: When an instruction causes an exception, we want to disable the updating of the programmer-visible state by later instructions and halt execution once the excepting instruction reaches the write-back stage.

Y86 PIPE: Pipeline Control Logic

```
0x000:    irmovl Stack,%esp  # Initialize stack pointer
0x006:    call Proc          # procedure call
0x00b:    irmovl $10,%edx    # return point
0x011:    halt
0x020: .pos 0x20
0x020: Proc:                 # Proc:
0x020:    ret                # return immediately
0x021:    rrmovl %edx,%ebx   # not executed
0x030: .pos 0x30
0x030: Stack:                # Stack: Stack pointer
```

Y86 PIPE: Pipeline Control Logic

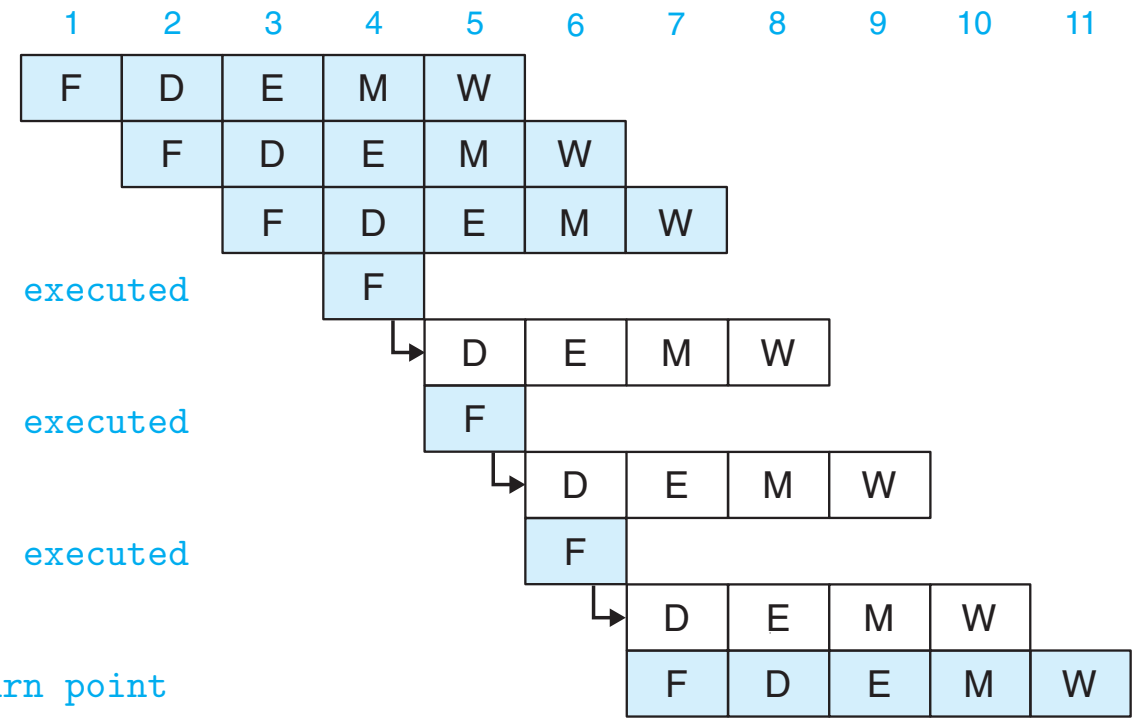
```
# prog7
0x000: irmovl Stack,%edx
0x006: call proc
0x020: ret
       bubble
       bubble
       bubble
0x00b: irmovl $10,%edx # Return point
```



Y86 PIPE: Pipeline Control Logic

Actual processing of the ret **instruction.**

```
# prog7
0x000: irmovl Stack,%edx
0x006: call proc
0x020: ret
0x021: rrmovl %edx,%ebx # Not executed
       bubble
0x021: rrmovl %edx,%ebx # Not executed
       bubble
0x021: rrmovl %edx,%ebx # Not executed
       bubble
0x00b: irmovl $10,%edx # Return point
```



Y86 PIPE: Pipeline Control Logic

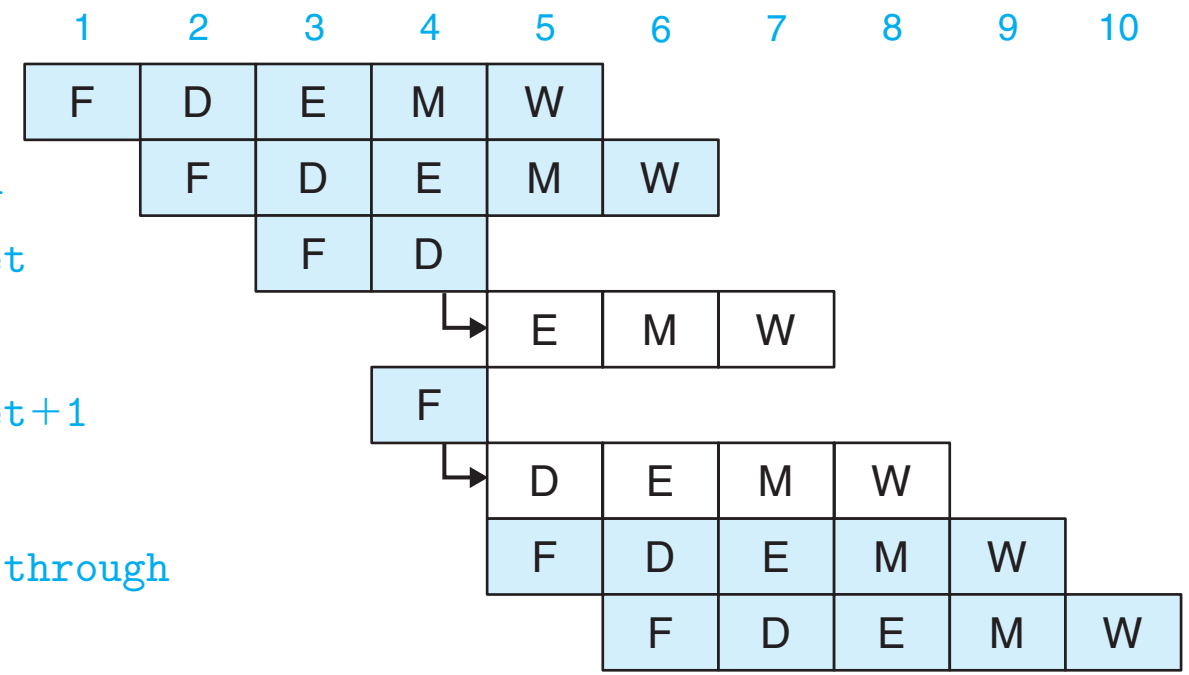
Processing mispredicted branch instructions

```
0x000:    xorl %eax,%eax
0x002:    jne  target          # Not taken
0x007:    irmovl $1, %eax      # Fall through
0x00d:    halt
0x00e: target:
0x00e:    irmovl $2, %edx      # Target
0x014:    irmovl $3, %ebx      # Target+1
0x01a:    halt
```

Y86 PIPE: Pipeline Control Logic

Processing mispredicted branch instructions

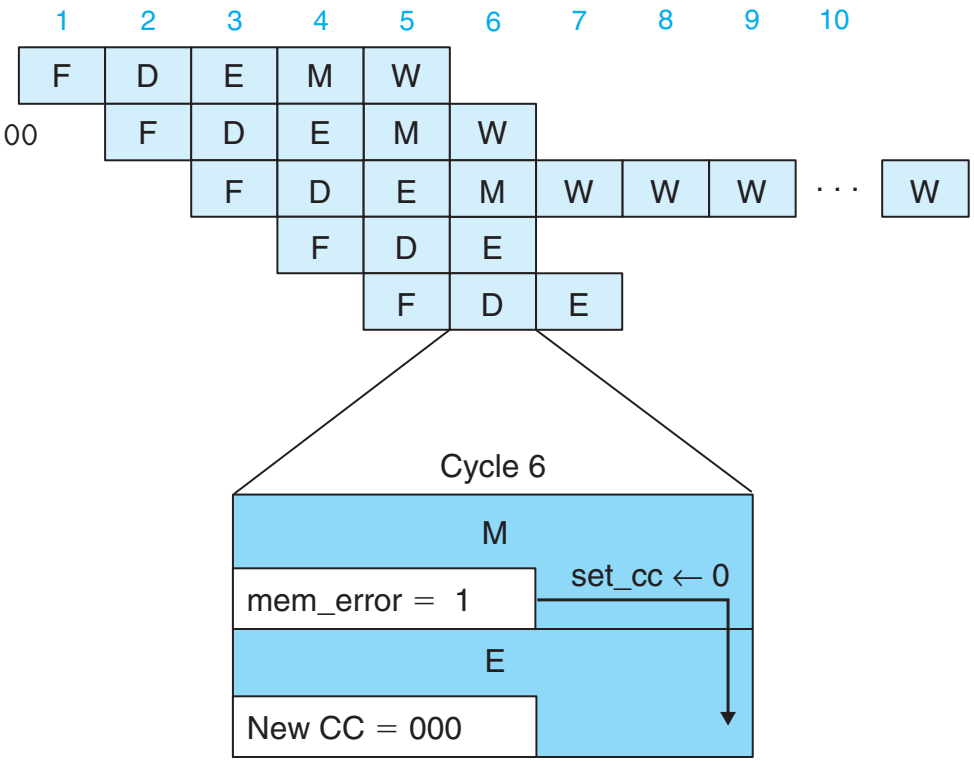
```
# prog8
0x000: xorl %eax,%eax
0x002: jne target # Not taken
0x00e: irmovl $2,%edx # Target
       bubble
0x014: irmovl $3,%ebx # Target+1
       bubble
0x007: irmovl $1,%eax # Fall through
0x00d: halt
```



Y86 PIPE: Pipeline Control Logic

Processing invalid memory reference exception

```
# prog10
0x000: irmovl $1,%eax
0x006: xorl %esp,%esp #CC = 100
0x008: pushl %eax
0x00a: addl %eax,%eax
0x00c: irmovl $2,%eax
```

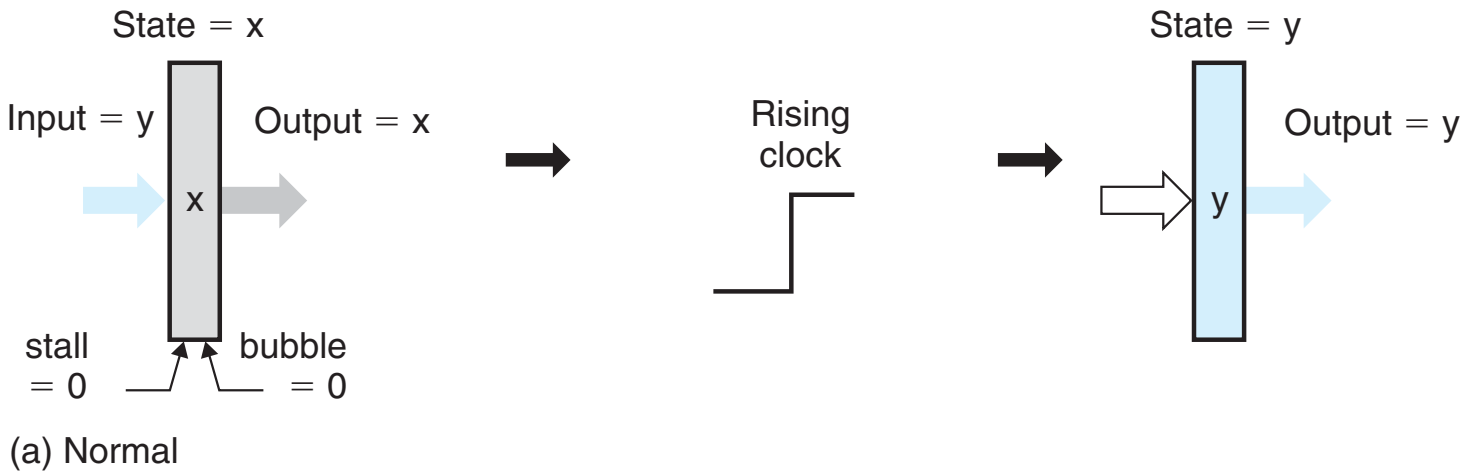


Y86 PIPE: Pipeline Control Logic

Detection conditions for pipeline control logic

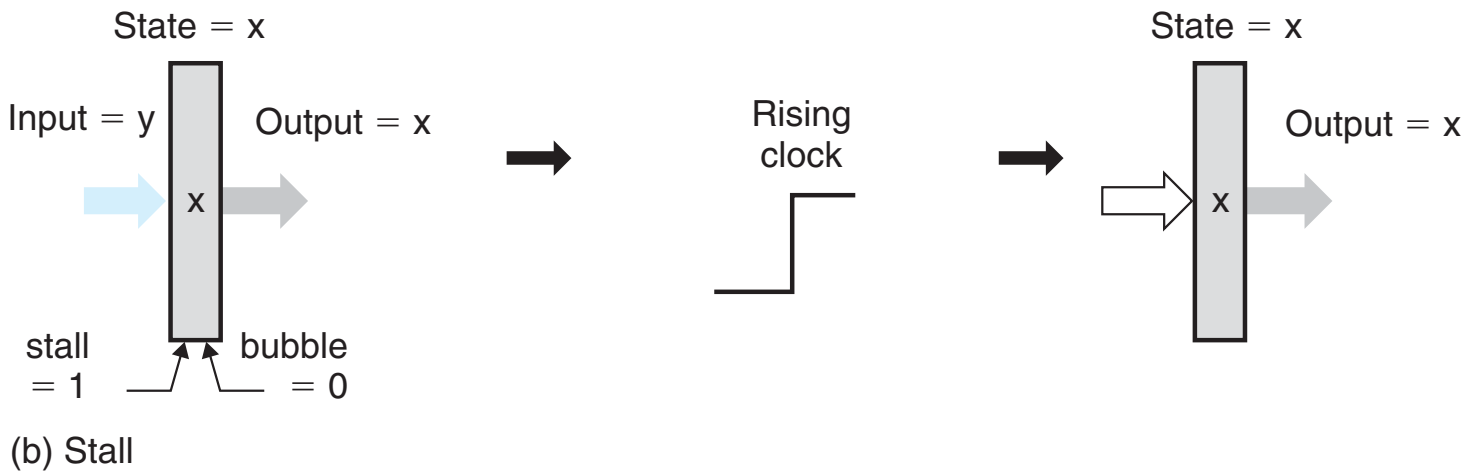
Condition	Trigger
Processing `ret`	IRET ∈ {D_icode, E_icode, M_icode}
Load/use hazard	E_icode ∈ {IMRMOVL, IPOPL} && E_dstM ∈ {d_srcA, d_srcB}
Mispredicted branch	E_icode = IJXX && !e_Cnd
Exception	m_stat ∈ {SADR, SINS, SHLT} || W_stat ∈ {SADR, SINS, SHLT}

Y86 PIPE: Pipeline Control Logic



- Under normal operation, both of these inputs are set to 0, causing the register to load its input as its new state.

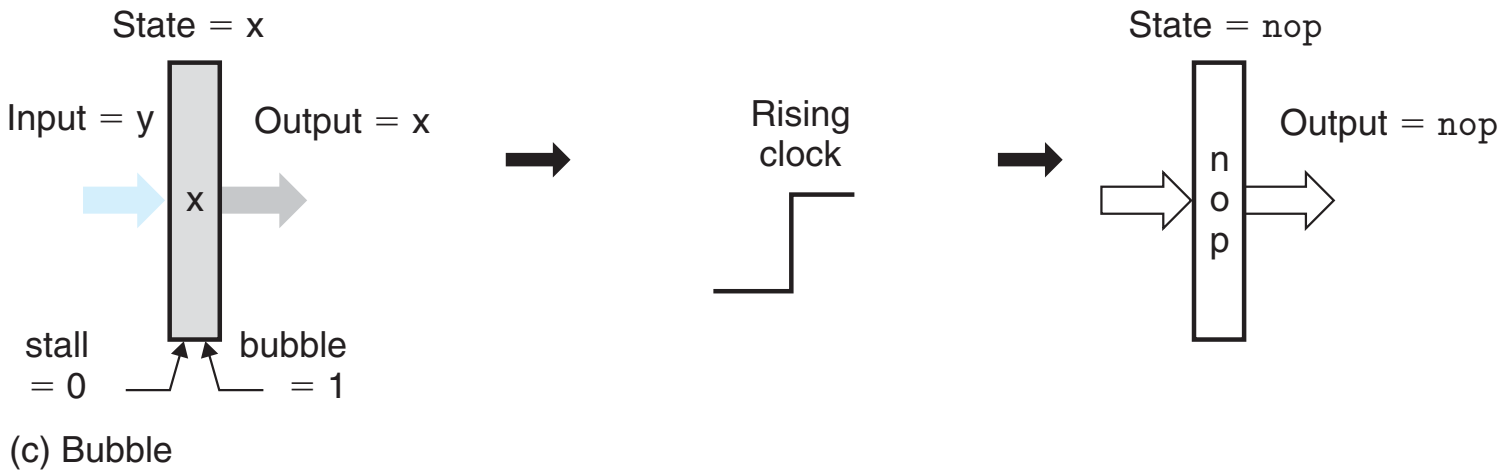
Y86 PIPE: Pipeline Control Logic



(b) Stall

- When the stall signal is set to 1, the updating of the state is disabled. Instead, the register will remain in its previous state.

Y86 PIPE: Pipeline Control Logic



(c) Bubble

- When the bubble signal is set to, the state of the register will be set to some fixed *reset configuration* giving a state equivalent to that of a nop instruction.

Y86 PIPE: Pipeline Control Logic

- When the bubble signal is set to, the state of the register will be set to some fixed *reset configuration* giving a state equivalent to that of a nop instruction.
    - To inject a bubble into pipeline register D, we want the icode field to be set to the constant value INOP.
    - 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.

Y86 PIPE: Pipeline Control Logic

- 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.

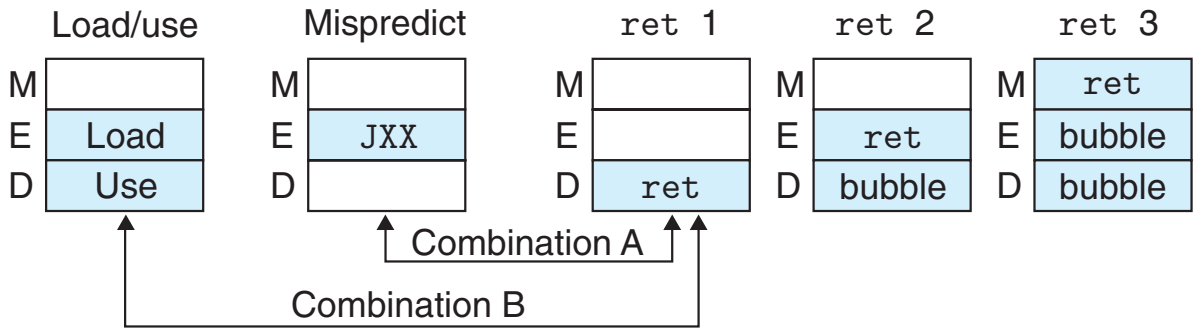
Y86 PIPE: Pipeline Control Logic

Combinations of Control Conditions :

	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

Y86 PIPE: Pipeline Control Logic

Combinations of Control Conditions :

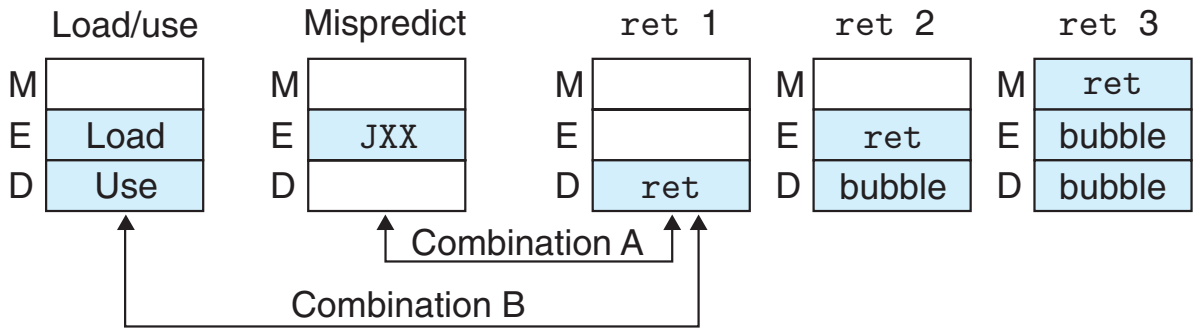


Most of the control conditions are mutually exclusive. For example, it is not possible to have a load/use hazard and a mispredicted branch simultaneously, since one requires a load instruction (mrmovl or popl) in the execute stage, while the other requires a jump.

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.

Y86 PIPE: Pipeline Control Logic

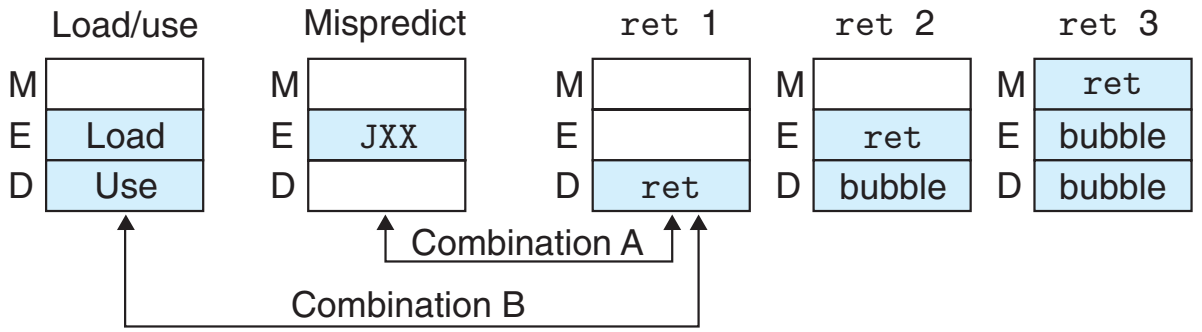
Combinations of Control Conditions :



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.

Y86 PIPE: Pipeline Control Logic

Combinations of Control Conditions :



Combination A: 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

Y86 PIPE: Pipeline Control Logic

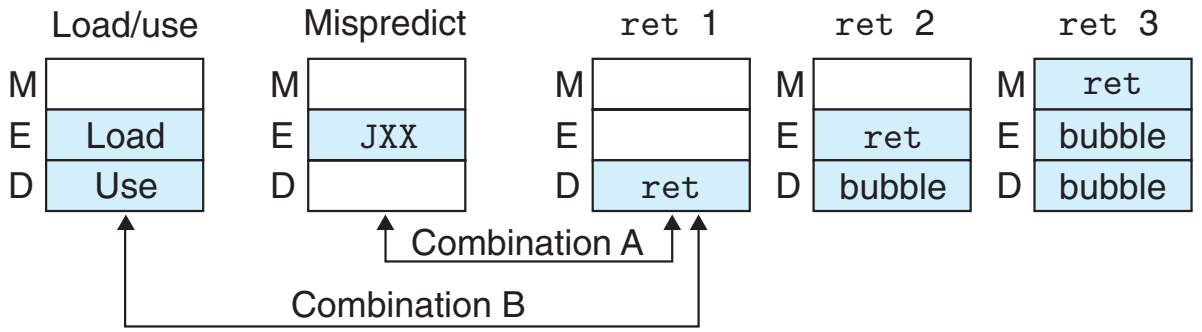
Combinations of Control Conditions :

	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.

Y86 PIPE: Pipeline Control Logic

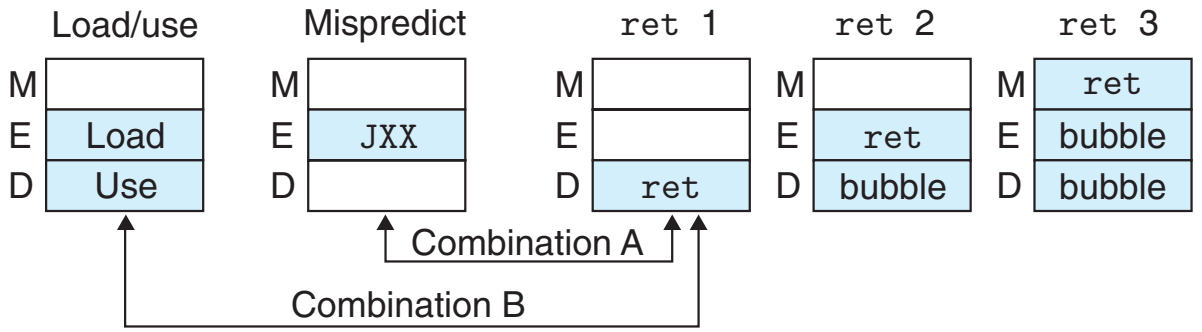
Combinations of Control Conditions :



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.

Y86 PIPE: Pipeline Control Logic

Combinations of Control Conditions :



Combination B:

	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

Y86 PIPE: Pipeline Control Logic

Combinations of Control Conditions :

	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 the pipeline to perform both sets of actions. Instead, we want it to 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.

Y86 PIPE: Pipeline Control Logic

Control Logic Implementation