Chapter 4: Y86 CPU Stages

Topics

- 6 Stages of Y86 CPU Instruction Execution
- Pipelining
- Data Hazards
- Branch Misprediction

Announcements

Buffer Lab is due next Monday Oct 27 by 8 am

- Note extension and note it is due by 8 am Monday
- Disregard Invalids on server scoreboard for now

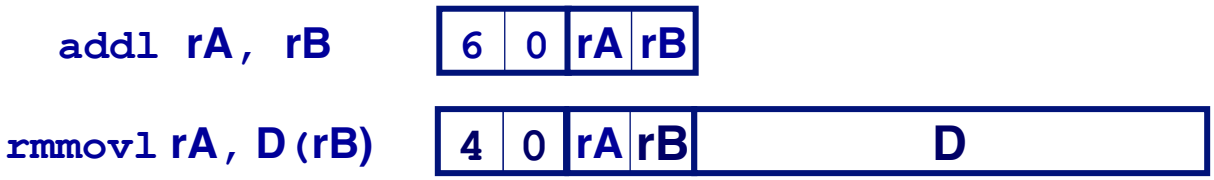
Essential that you read the textbook in detail & do the practice problems

- Read Chapter 4, but Skip 4.2, 4.3.4, 4.5.9-4.5.11 (skip the PIPE implementation), 4.5.13. Overall, skipping these sections will save you about 50 pages of reading

Recap…

Y86 ISA

- Subset of X86 32-bit ISA for instructional purposes
  - Study how a single instruction like `addl` is actually executed
- Limited # of instructions
  - `addl, subl, andl, xorl1, jXX, pushl, popl, call, ret`
  - `irmovl, rrmovl, mrmovl, rmmovl`
- Instruction coding is a variable # of bytes



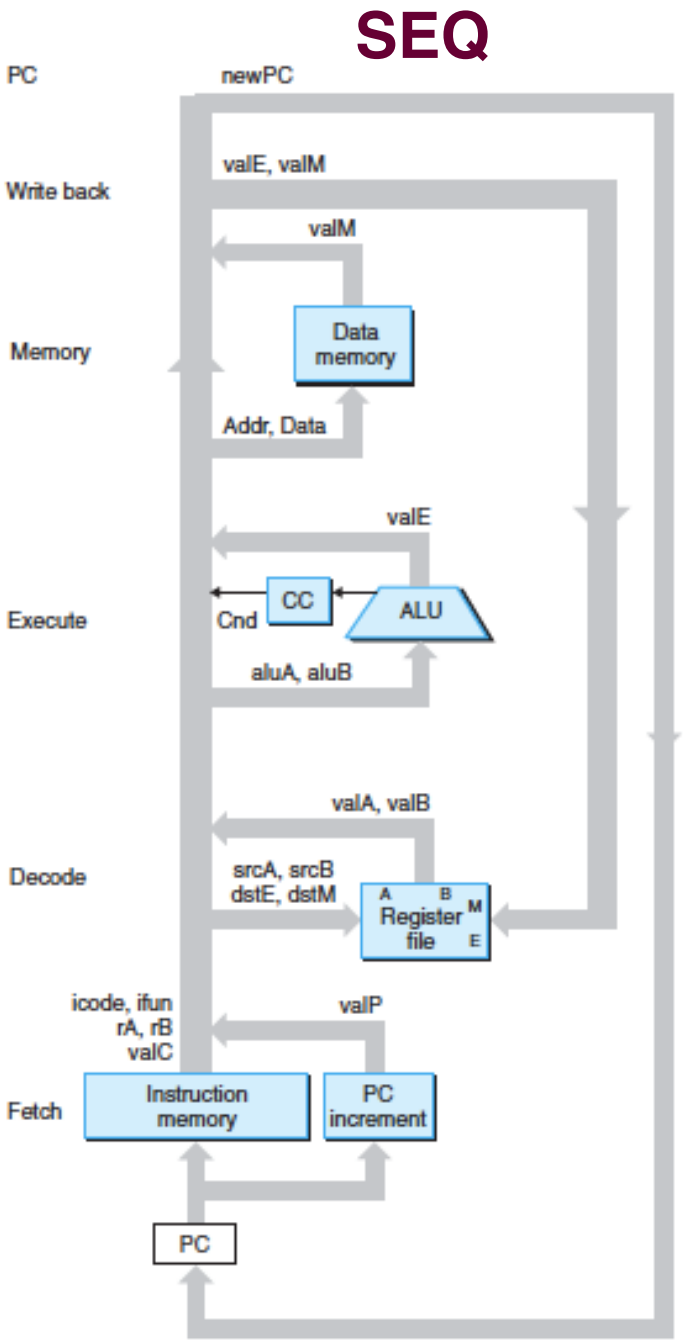
Recap…

Y86 assembly program example

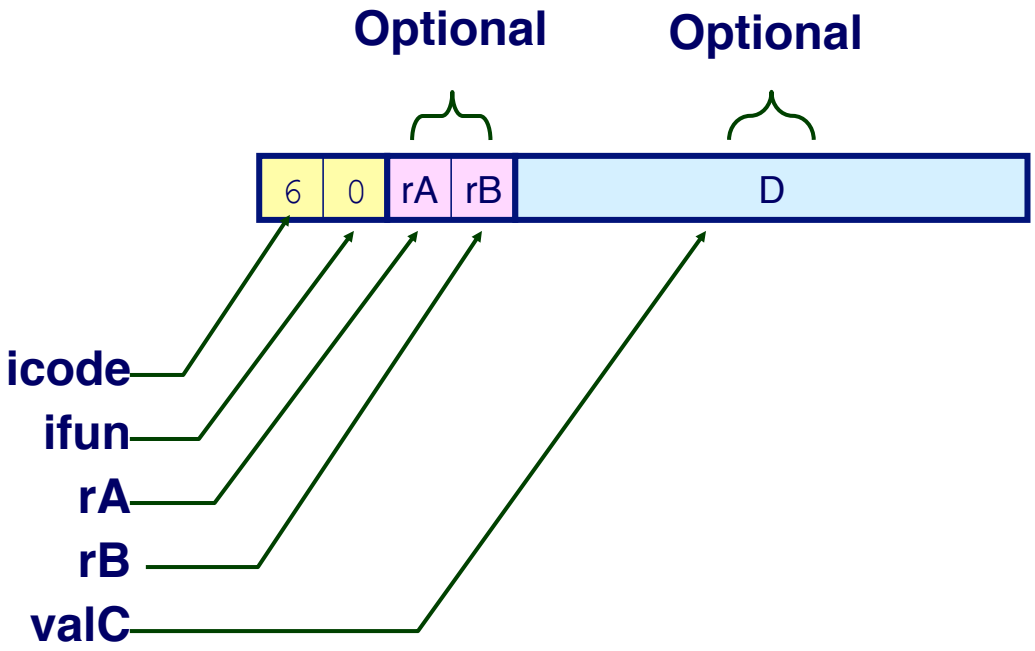
- movl -> irmovl, rrmovl, rmmovl, mrmovl
- incl and addl $1, * -> two instructions
- Use simplified addressing instead of complex addressing modes
- Each instruction is fundamentally basic, and you can build more complex instructions by combining basic ones

Y86 CPU instruction execution

- 6 Stages: Fetch, Decode, Execute, Memory, Write Back, PC Update
- This is the basic “SEQ” Architecture of the textbook
- Note: the memory is not actually inside the CPU as pictured



Instruction Decoding



Instruction Format – Different Fields

- Instruction byte — icode:ifun
- Optional register byte — rA:rB
- Optional constant word — valC

Executing An Arithmetic/Logical Operation

`OPl rA, rB` 

Fetch
- Read 2 bytes

Decode
- Read operand registers

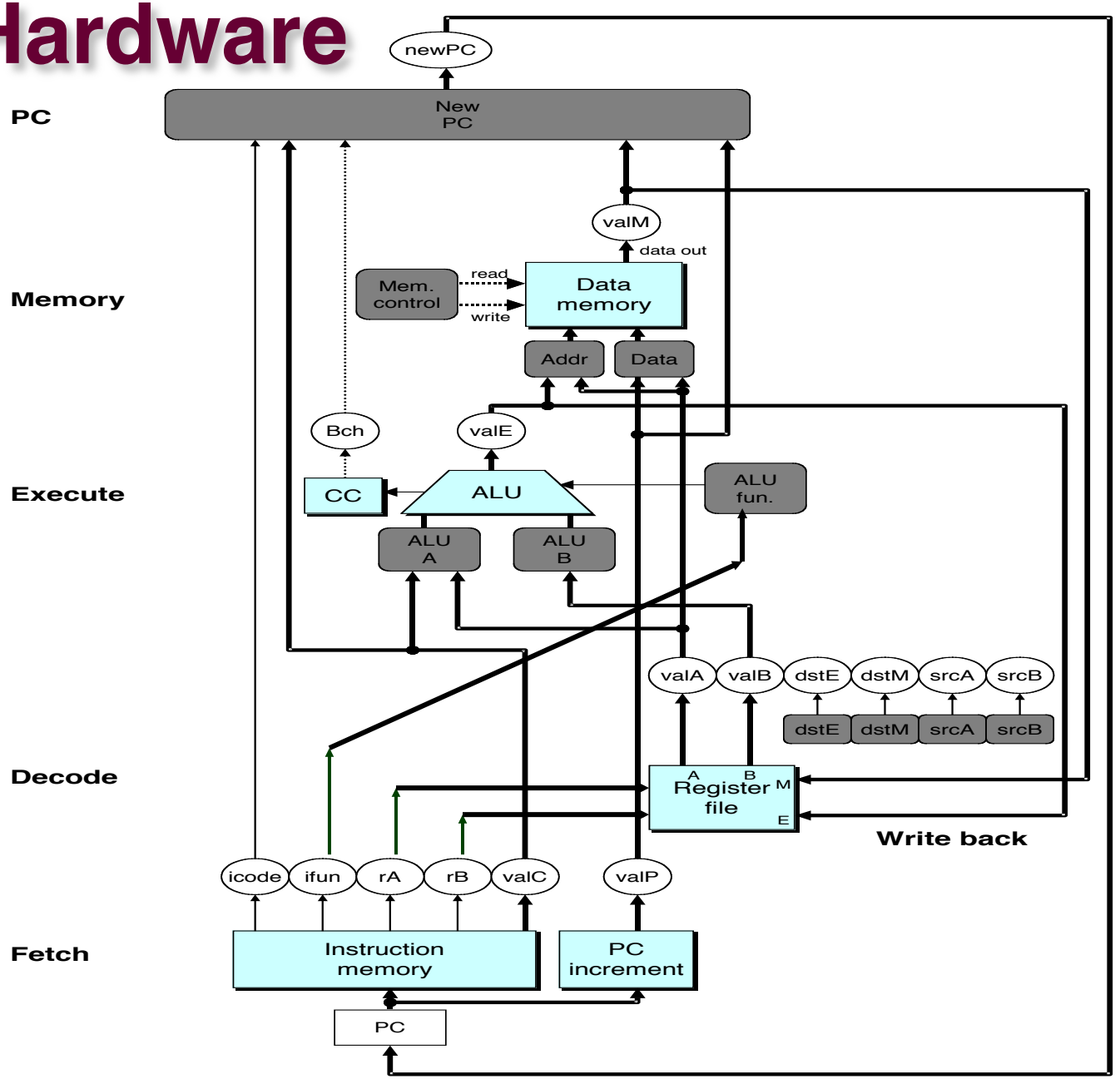
Execute
- Perform operation
- Set condition codes

Memory
- Do nothing

Write back
- Update register

PC Update
- Increment PC by 2

SEQ Hardware



Some lines omitted for brevity

Stage Computation: Arith/Log. Ops

	OPl rA, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA ← R[rA] valB ← R[rB]	Read operand A Read operand B
Execute	valE ← valB OP valA Set CC	Perform ALU operation Set condition code register
Memory		
Write back	R[rB] ← valE	Write back result
PC update	PC ← valP	Update PC

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions

Execution Example

addl %edx, %ecx

- Encoding: 0x6021
- Byte 1
  - icode=6
  - ifun=0
- Byte 2
  - rA=%edx=2
  - rB=%ecx=1

PC

5. Memory stage is unused for Y86 addl

Memory

4. Result is written back to %ecx.

Execute

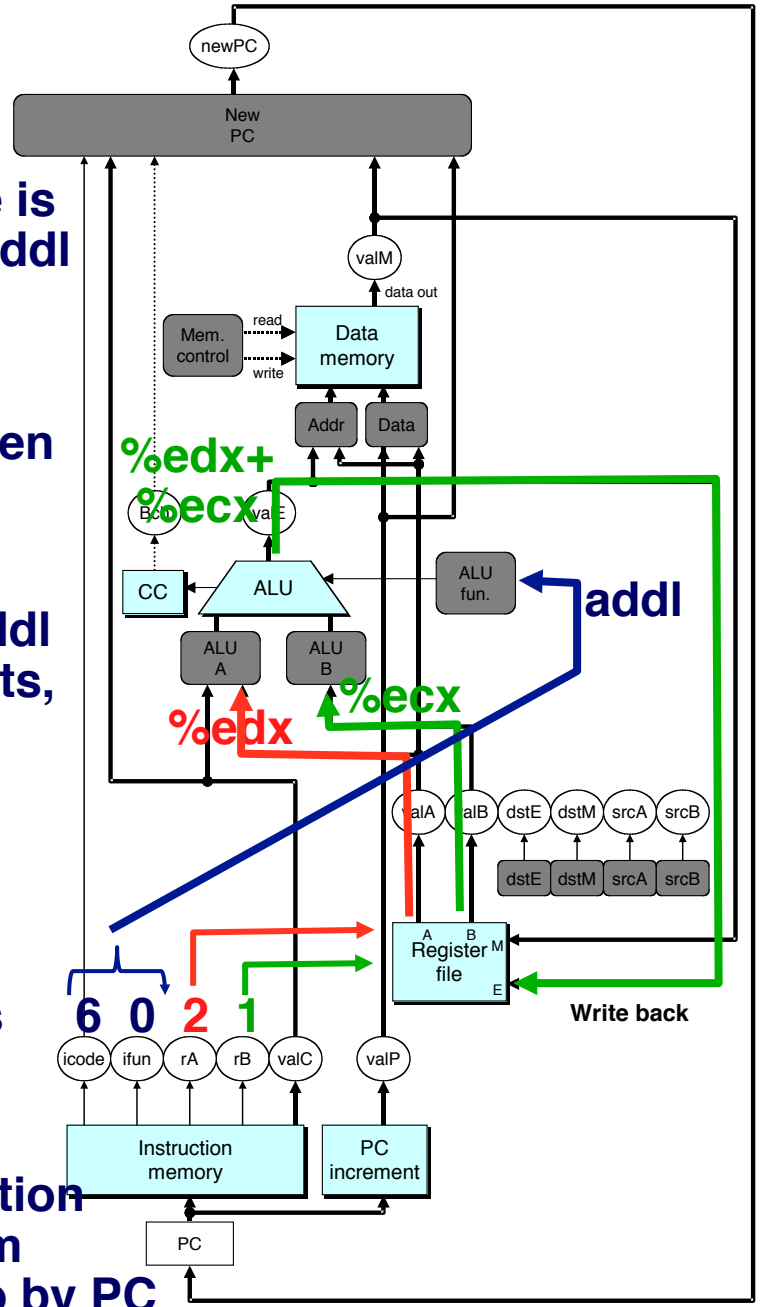
3. ALU executes addl instruction on inputs, generates result

Decode

2. Decode values

Fetch

1. Fetch instruction = 0x6021 from mem loc pointed to by PC



Executing `rmmovl`

Fetch

- Read 6 bytes

Decode

- Read operand registers

Execute

- Compute effective address

Memory

- Write to memory

Write back

- Do nothing

PC Update

- Increment PC by 6

Stage Computation: `rmmovl`

	`rmmovl` rA, D(rB)	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_4[PC+2]	Read displacement D
	valP ← PC+6	Compute next PC
Decode	valA ← R[rA]	Read operand A
	valB ← R[rB]	Read operand B
Execute	valE ← valB + valC	Compute effective address
Memory	M_4[valE] ← valA	Write value to memory
Write back		
PC update	PC ← valP	Update PC

- Use ALU for address computation

rmmovl Execution

Example:

rmmovl %edx, 8(%ecx)

- Encoding: 0x40210800000 0
- Byte 1
  - icode=4
  - ifun=0
- Byte 2
  - rA=%edx=2
  - rB=%ecx=1
- Bytes 3-6
  - valC = 8 = Displacement D
  - Note D is stored in byte-reversed little Endian order = 0x08000000

PC

5. No writeback stage for rmmovl

Memory

4. %edx is written in memory location %ecx+8

Execute

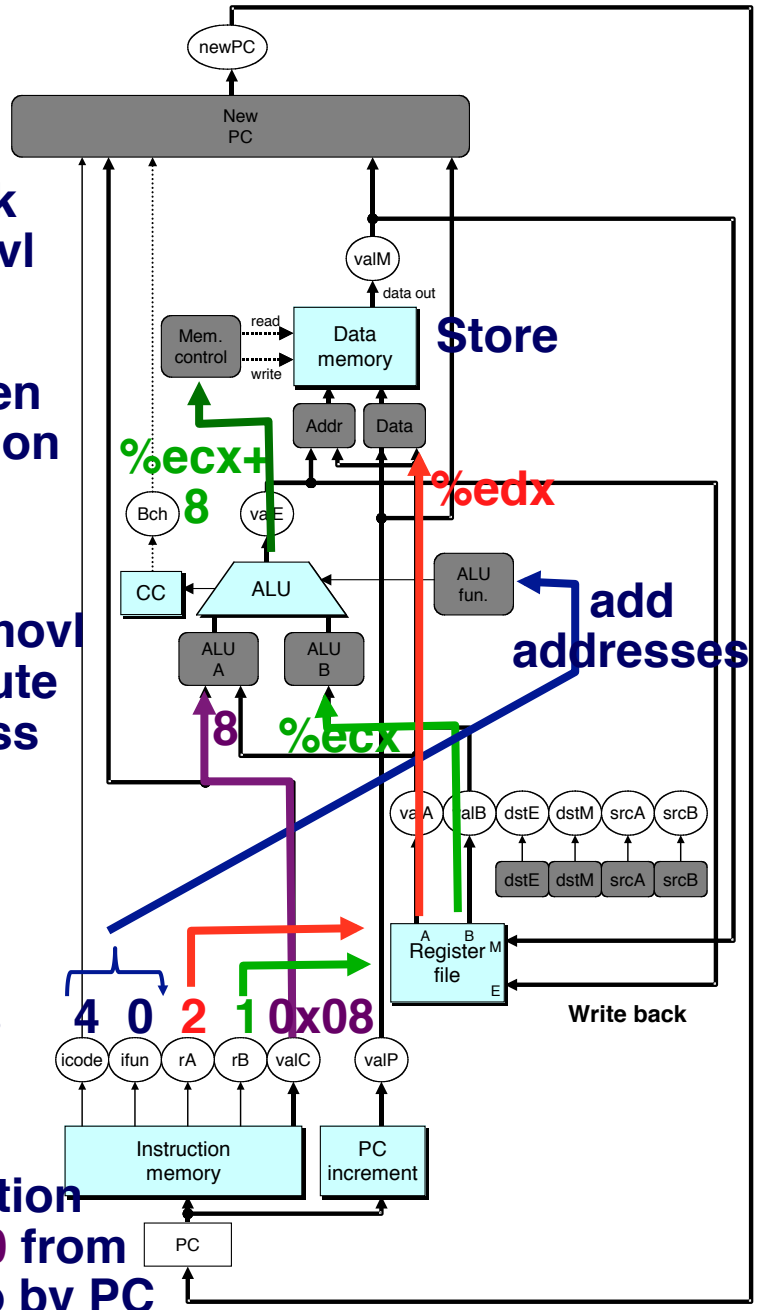
3. ALU executes rmmovl by adding to compute an effective address

Decode

2. Decode values 4 0 2 1 0x08

Fetch

1. Fetch instruction = 0x40210800000 from mem loc pointed to by PC



Executing `popl`

Fetch

- Read 2 bytes

Decode

- Read stack pointer

Execute

- Increment stack pointer by 4

Memory

- Read from old stack pointer

Write back

- Update stack pointer
- Write result to register

PC Update

- Increment PC by 2

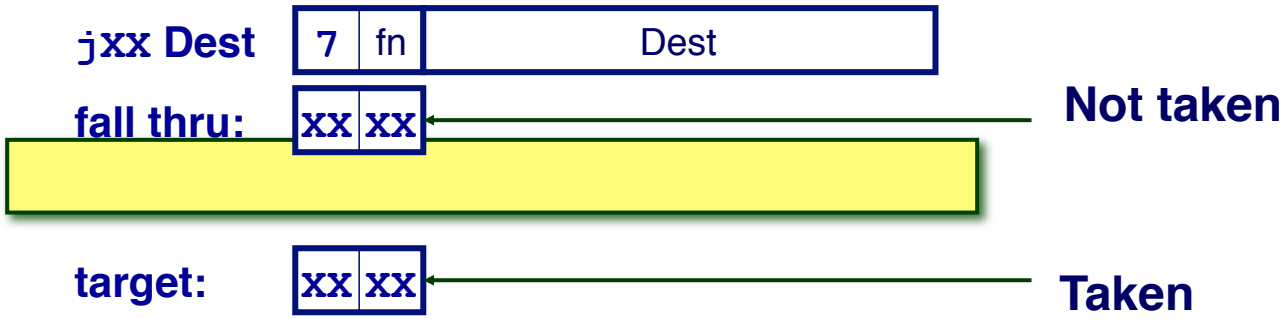
- Normally you first pop into a register, then shrink the stack pointer, but here we reverse the order due to the hardware stage design
  - Have to remember both old and new stack pointer values

Stage Computation: `popl`

	`popl` rA	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valP ← PC+2	Compute next PC
Decode	valA ← R[`%esp`]	Read stack pointer
	valB ← R [`%esp`]	Read stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M_4[valA]	Read from stack
Write back	R[`%esp`] ← valE	Update stack pointer
	R[rA] ← valM	Write back result
PC update	PC ← valP	Update PC

- Use ALU to increment stack pointer
- Must update two registers
  - Popped value
  - New stack pointer

Executing Jumps



Fetch

- Read 5 bytes
- Increment PC by 5

Decode

- Do nothing

Execute

- Determine whether to take branch based on jump condition and condition codes

Memory

- Do nothing

Write back

- Do nothing

PC Update

- Set PC to Dest if branch taken or to incremented PC if not branch

Stage Computation: Jumps

	jXX Dest	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_4[PC+1]	Read destination address
	valP ← PC+5	Fall through address
Decode		
Execute	Bch ← Cond(CC,ifun)	Take branch?
Memory		
Write back		
PC update	PC ← Bch ? valC : valP	Update PC

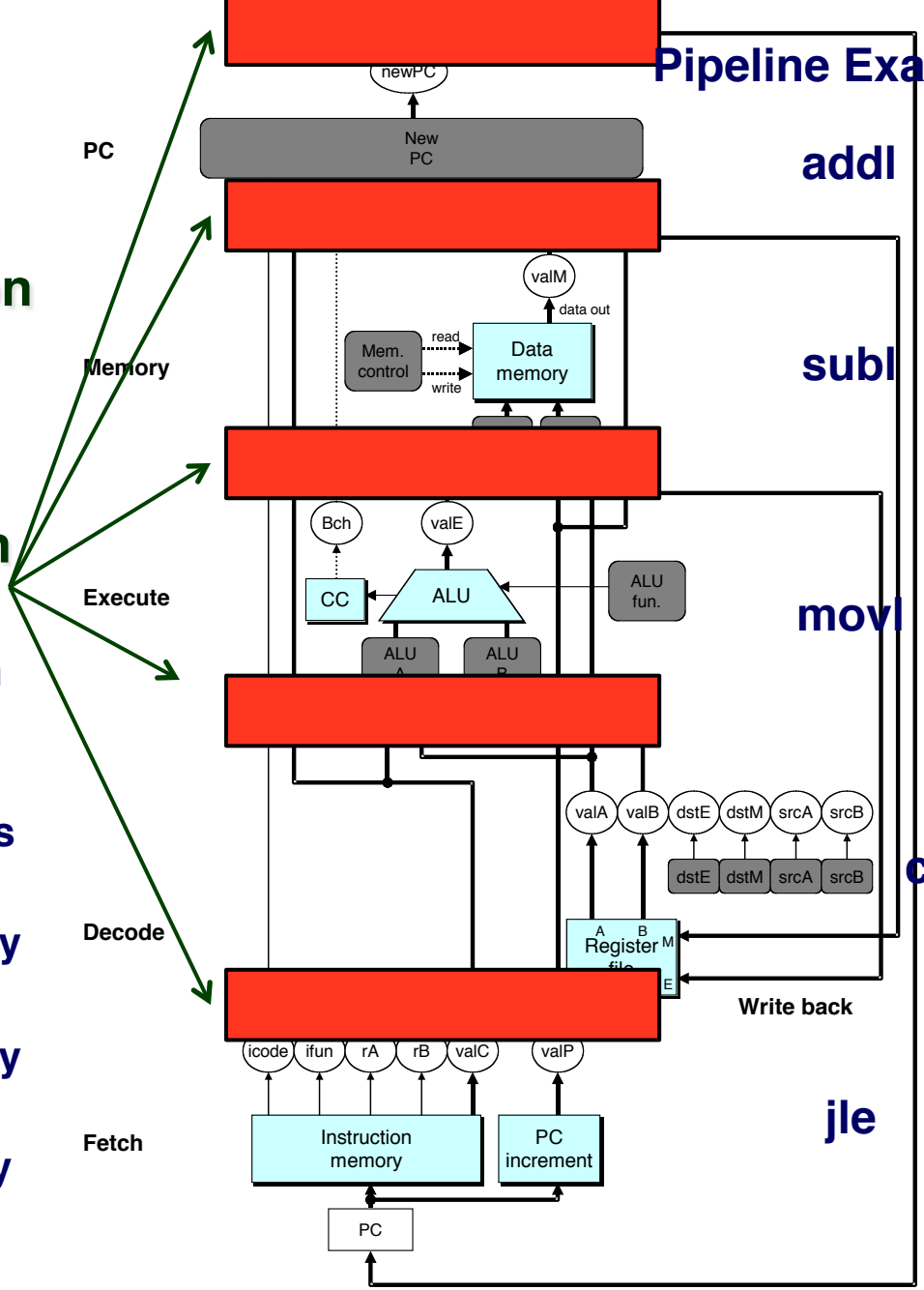
- Compute both addresses
- Choose based on setting of condition codes and branch condition

Pipelining

Executing a single instruction at a time leaves many stages idle

What happens if we insert a bank of registers between each stage?

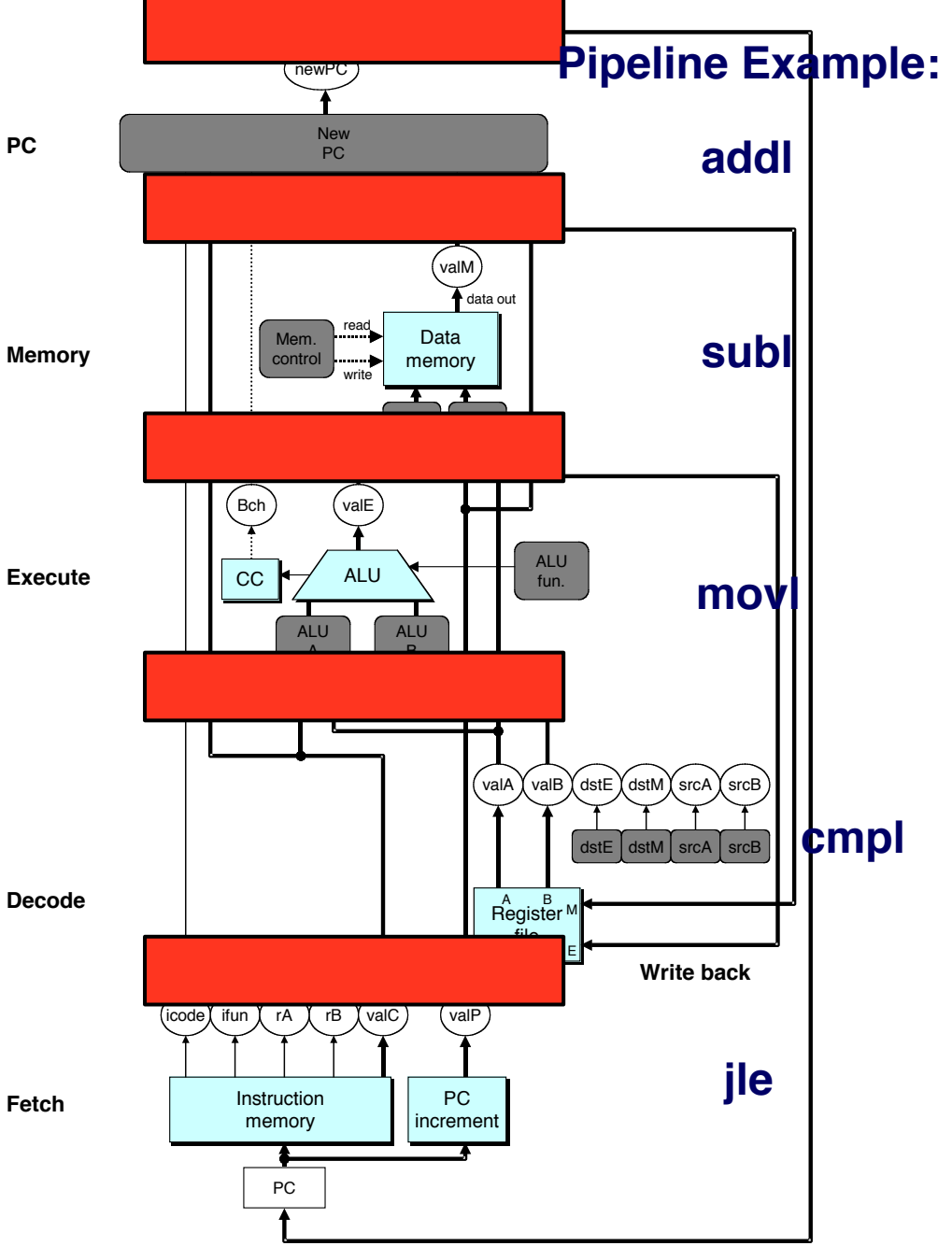
- **The result of each stage can be saved**
- **As instruction I reaches the Writeback stage, all previous stages are busy**
  - **Memory stage occupied by instructions I+1**
  - **Execute stage occupied by instruction I+2**
  - **Decode stage occupied by instruction I+3**
  - **Fetch stage occupied by instruction I+4, .etc…**



Pipelining

End result is faster *rate* of execution thanks to pipelining

- Previous non-pipelined rate = one instruction per all six stages
- New pipelined rate = one instruction per a single stage, i.e. interarrival time between instructions at output of pipeline is one stage
- New pipelined rate >> previous non-pipelined rate
- However, the cumulative delay of each individual instruction through the full CPU (all stages) is still approximately the same



Real-World Pipelines: Car Washes

Sequential

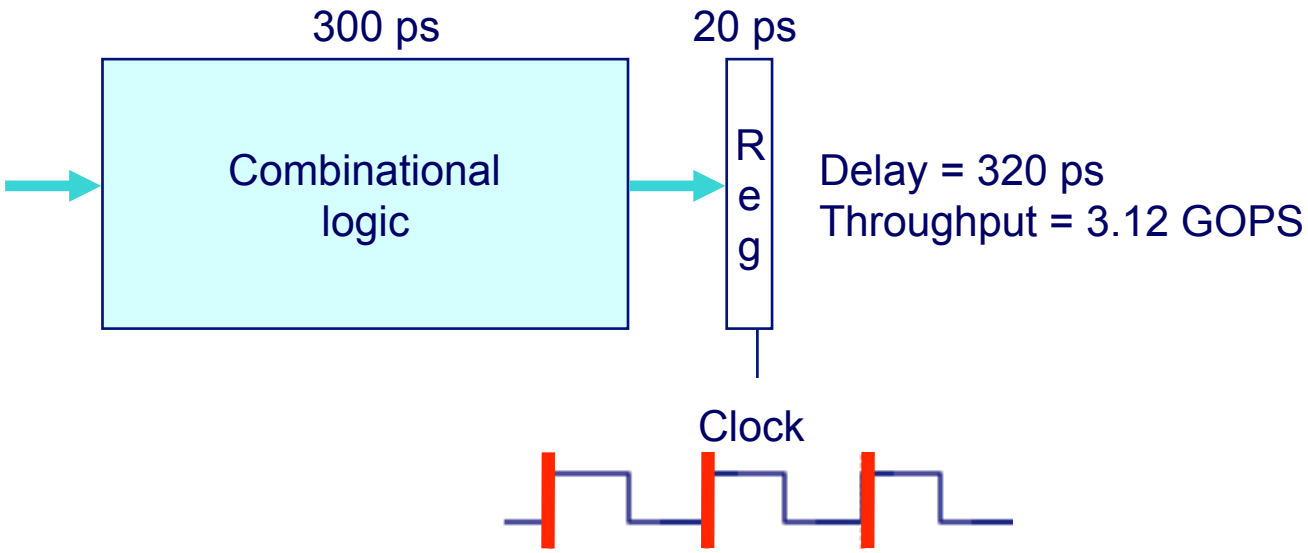
Parallel

Pipelined

Idea

- Divide process into independent stages
- Move objects through stages in sequence
- At any given time, multiple objects being processed

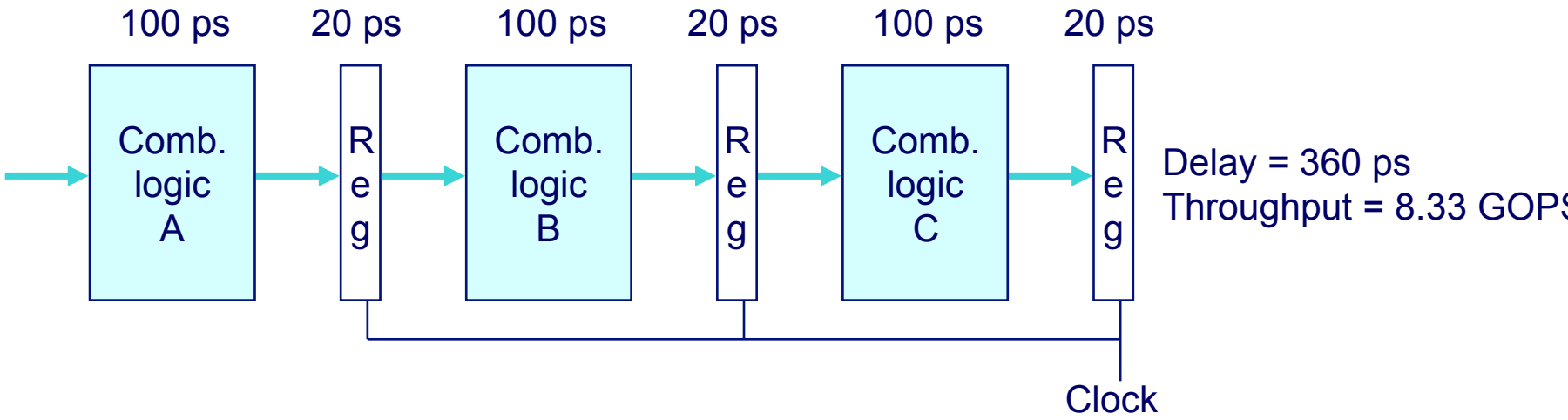
Computational Example



System

- Register stores or latches the results of the combinational logic on the *rising edge* of the clock
- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Must have clock cycle of at least 320 ps
- If combinational logic represents a CPU processing an instruction, then throughput = 1/320 ps = 3.12 GOPS

3-Way Pipelined Version

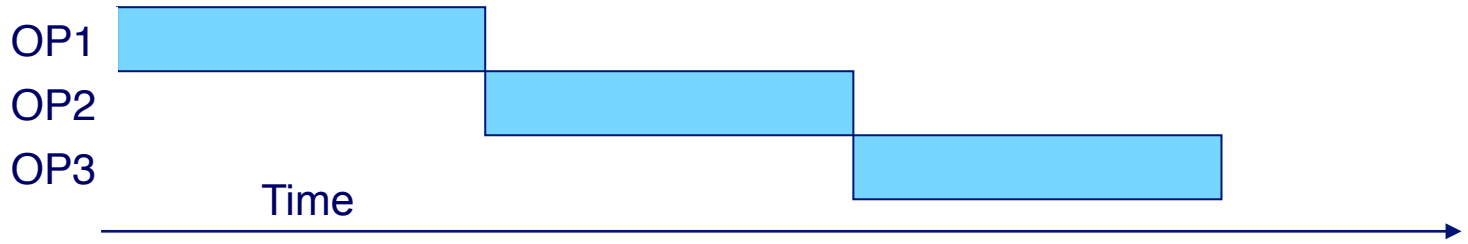


System

- Divide combinational logic into 3 blocks of 100 ps each
- Can begin new operation as soon as previous one passes through stage A.
  - Begin new operation every 120 ps
  - Throughput = 1/120 ps = 8.33 GOPS ← 2.67 times faster throughput!
- Overall latency increases
  - 360 ps from start to finish

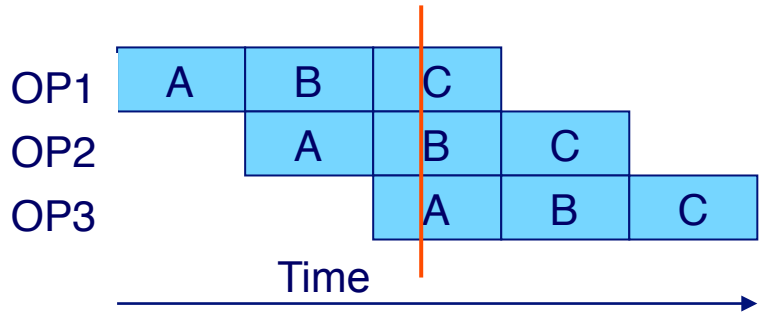
Pipeline Diagrams

Unpipelined



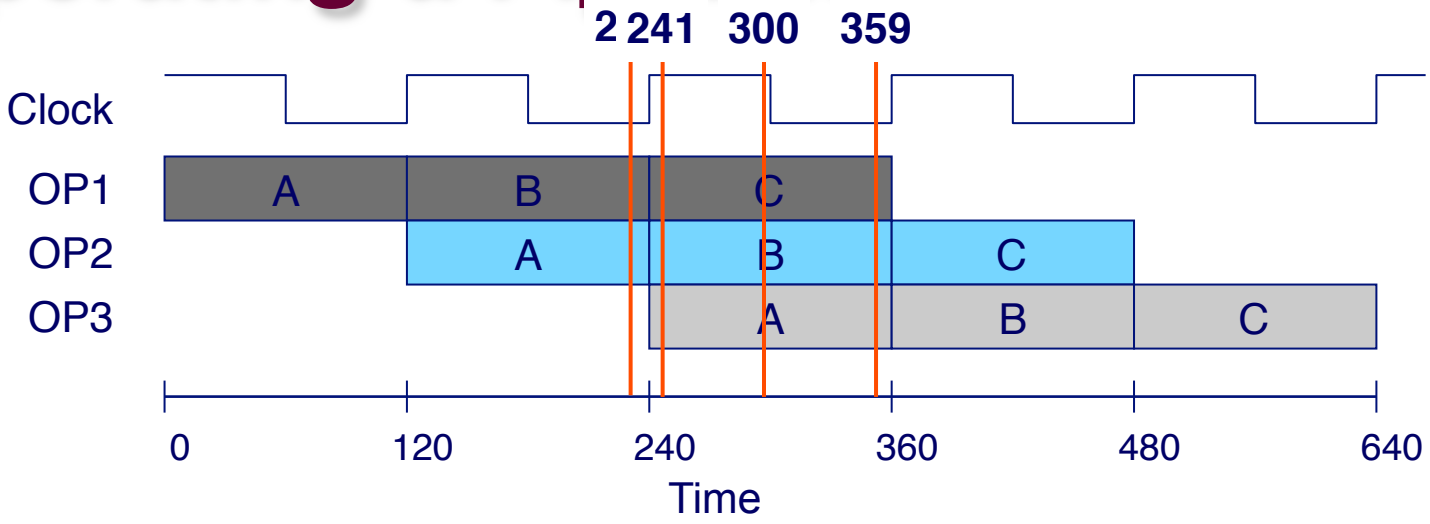
- Cannot start new operation until previous one completes

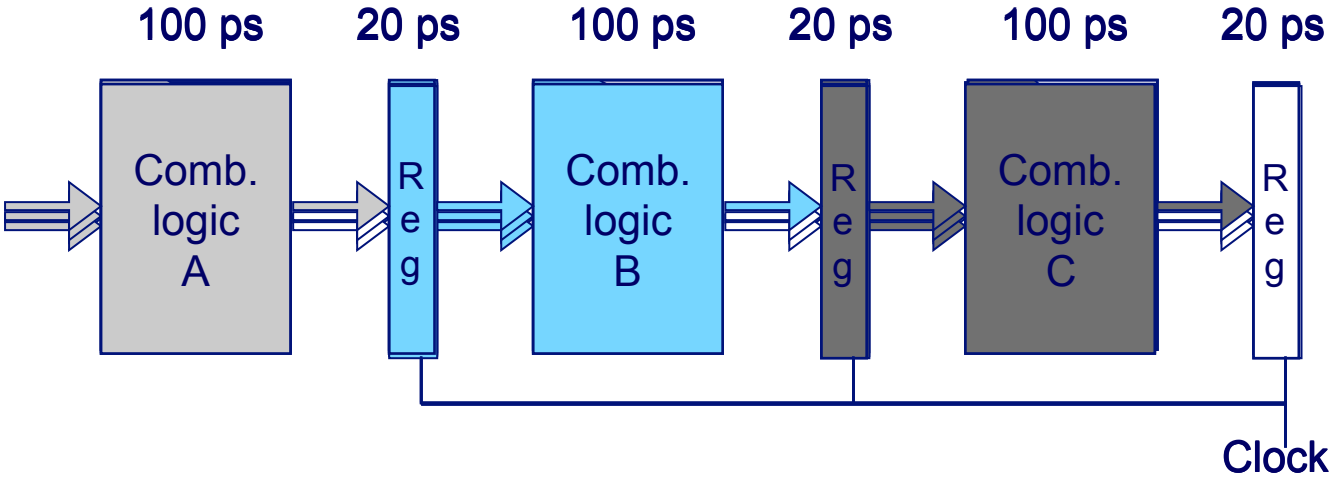
3-Way Pipelined



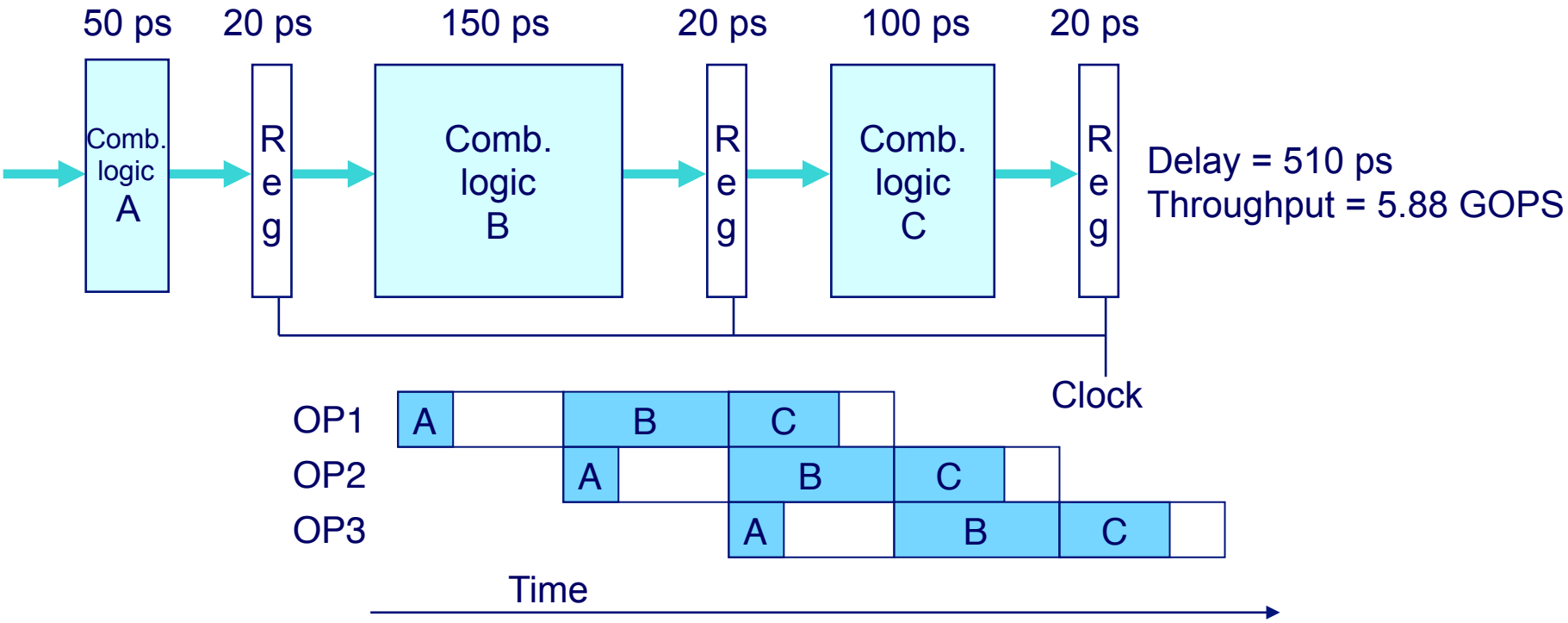
- Up to 3 operations in process simultaneously

Operating a Pipeline



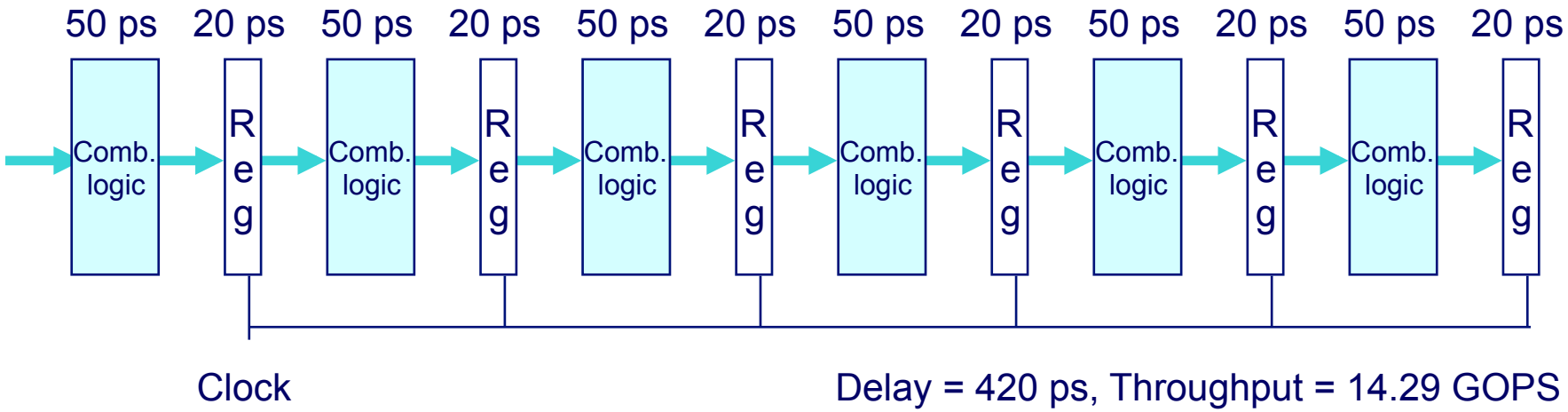


Limitations: Nonuniform Delays



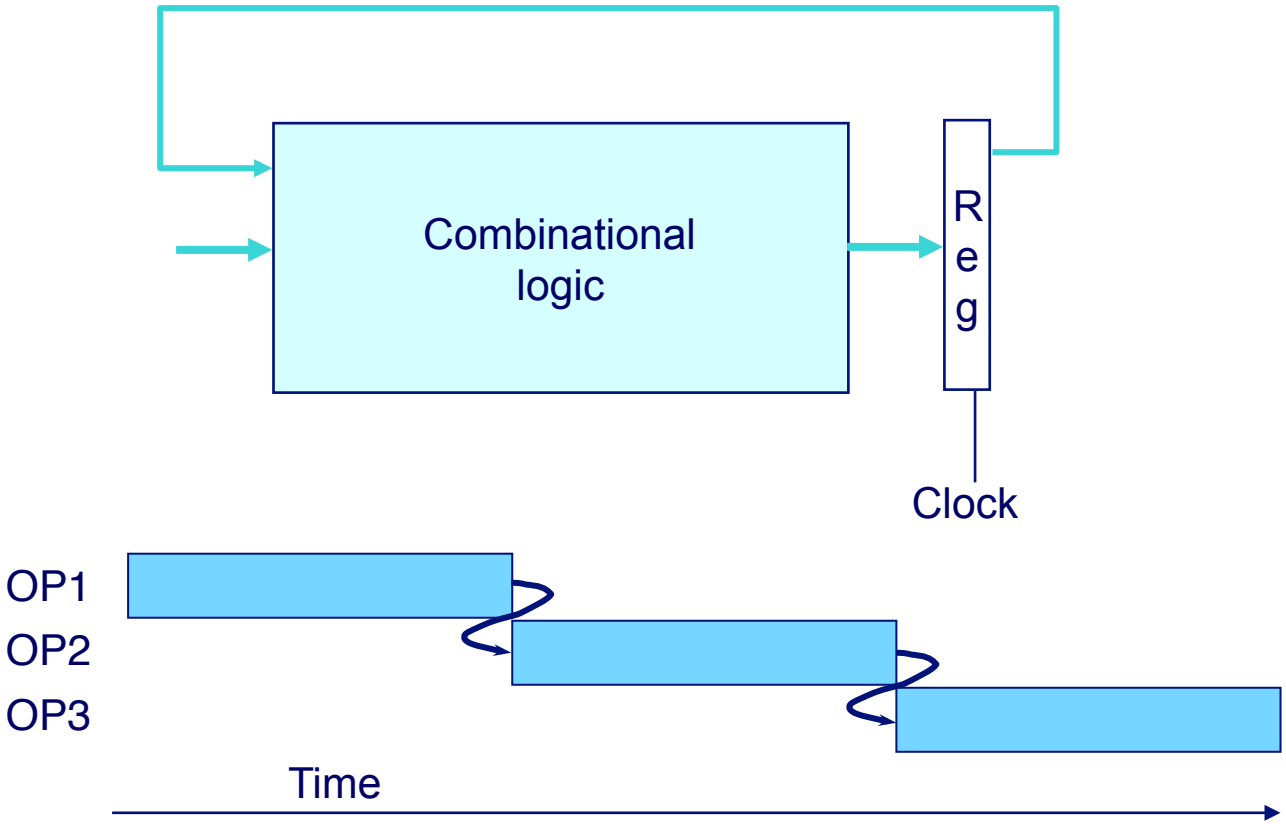
- **Throughput limited by slowest stage**
- **Other stages sit idle for much of the time**
- **Challenging to partition system into balanced stages**

Limitations: Register Overhead



- As we try to deepen the pipeline, the overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining (15 or more stages)

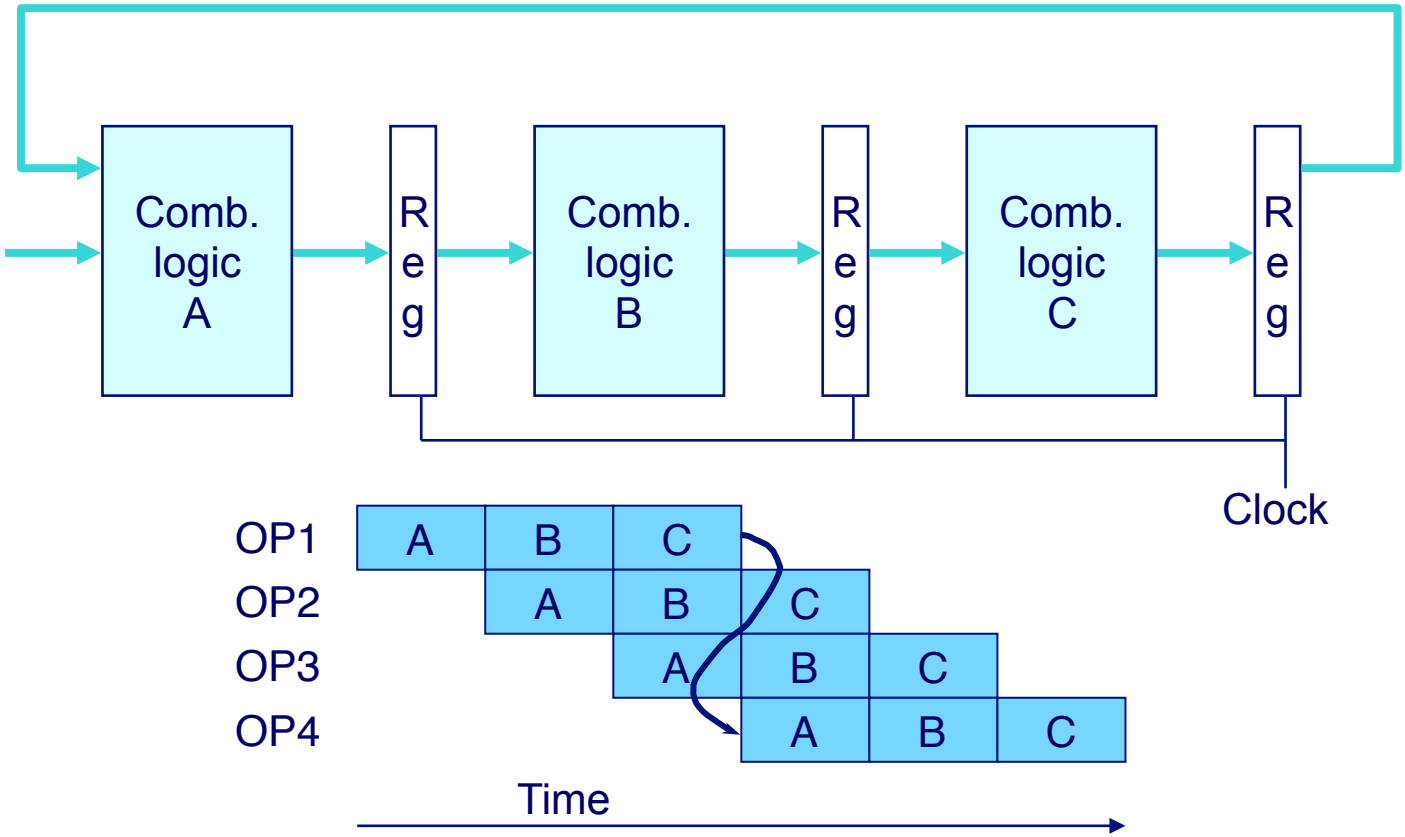
Data Dependencies



System

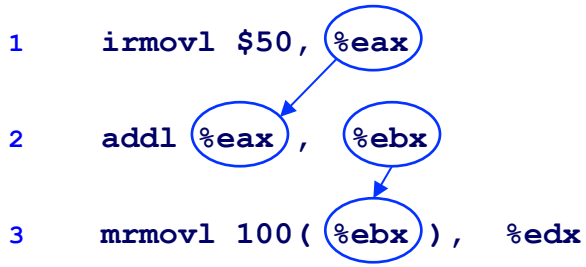
- **Each operation depends on result from preceding one**

Data Hazards



- **Result does not feed back around in time for next operation**
- **Pipelining has changed behavior of system**

Data Dependencies in Processors

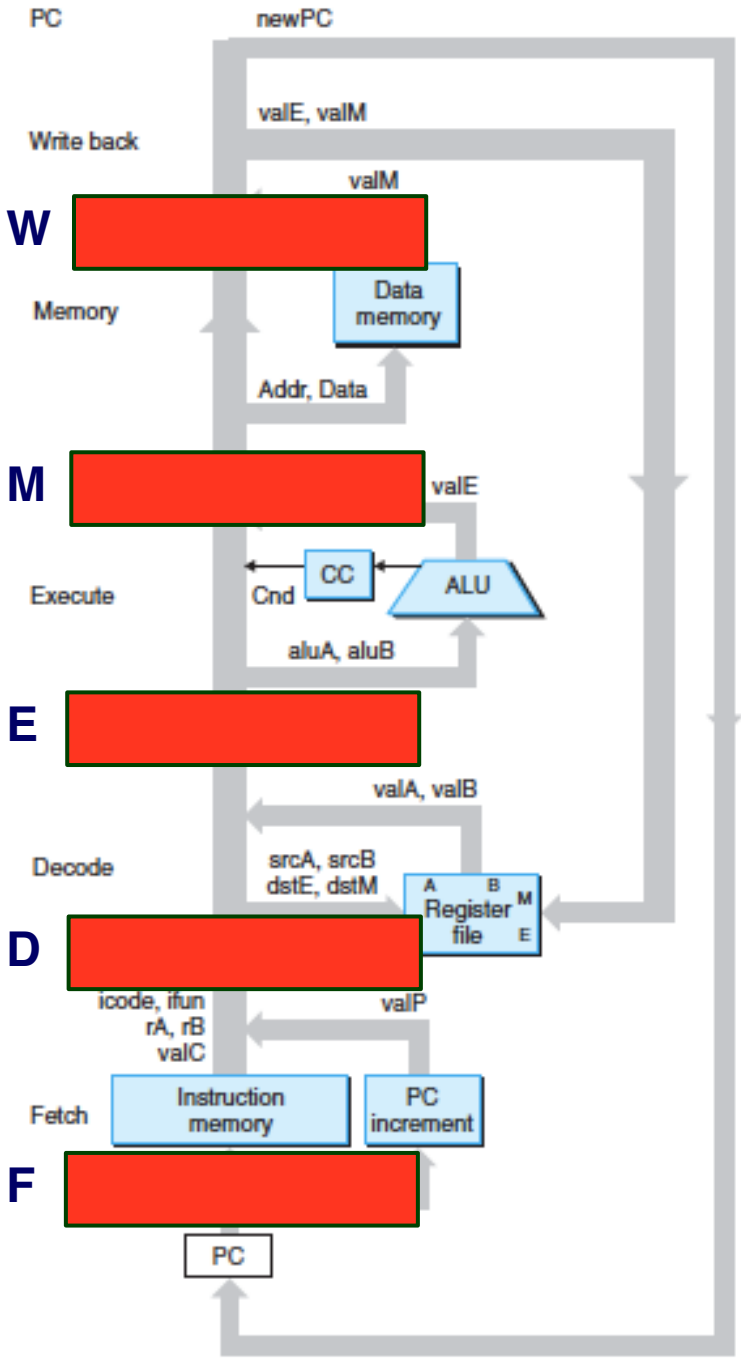


- Result from one instruction used as operand for another
  - Read-after-write (RAW) dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

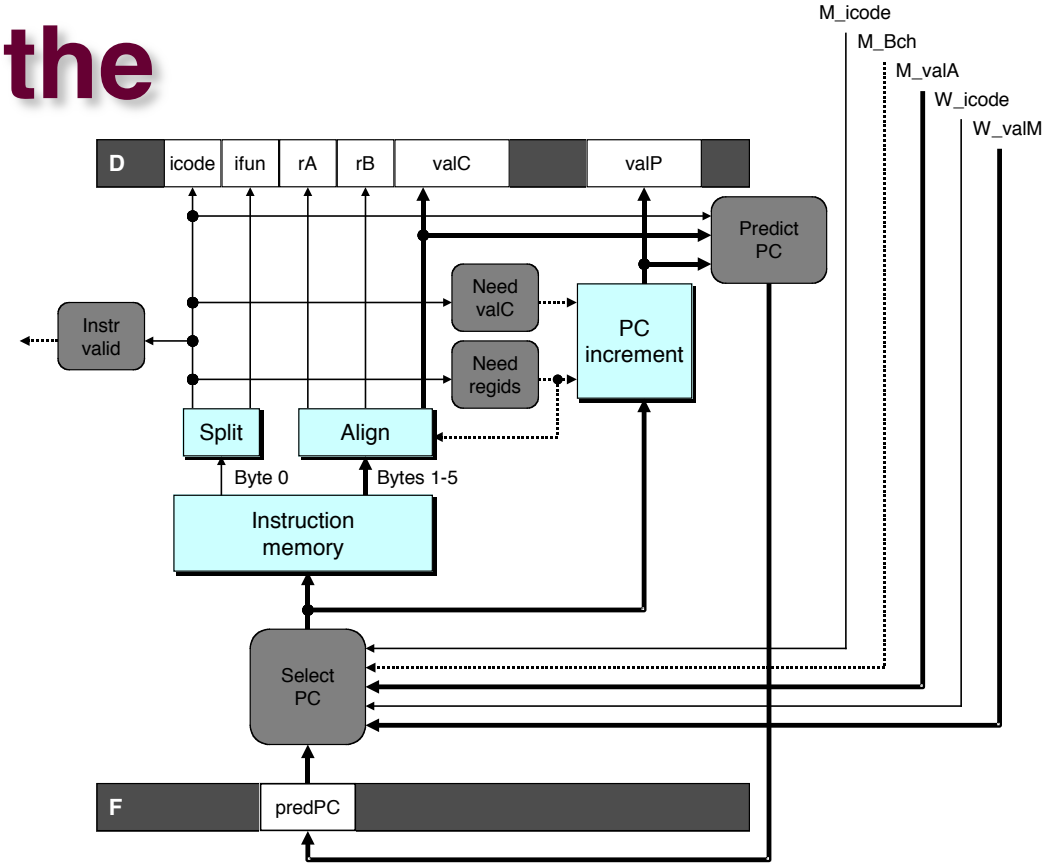
Adding Pipeline Registers

Add pipeline registers between each stage

- W registers are between the Memory and Writeback stages
- M registers are between the Execute and Memory stages
- E registers are between the Decode and Execute stages
- D registers are between Fetch and Decode stages
- F registers are for storing the predicted next instruction of the program counter PC
  - Reorganize Y86 CPU to do PC prediction in the first stage rather than the last stage - see textbook



Predicting the PC



- Start fetch of new instruction after current one has completed fetch stage
  - Not enough time to reliably determine next instruction
- Guess which instruction will follow
  - Recover if prediction was incorrect

Our PC Prediction Strategy

Instructions that Don't Transfer Control

- **Examples: sequence of instructions like `addl` followed by `irmovl` followed by `andl` followed by …**
- **Predict next PC to be valP**
- **Always reliable**

Call and Unconditional Jumps:

- **e.g. `call/jmp` 0xffff0a42**
- **Predict next PC to be valC (destination)**
- **Always reliable**

Our PC Prediction Strategy

Conditional Jumps:

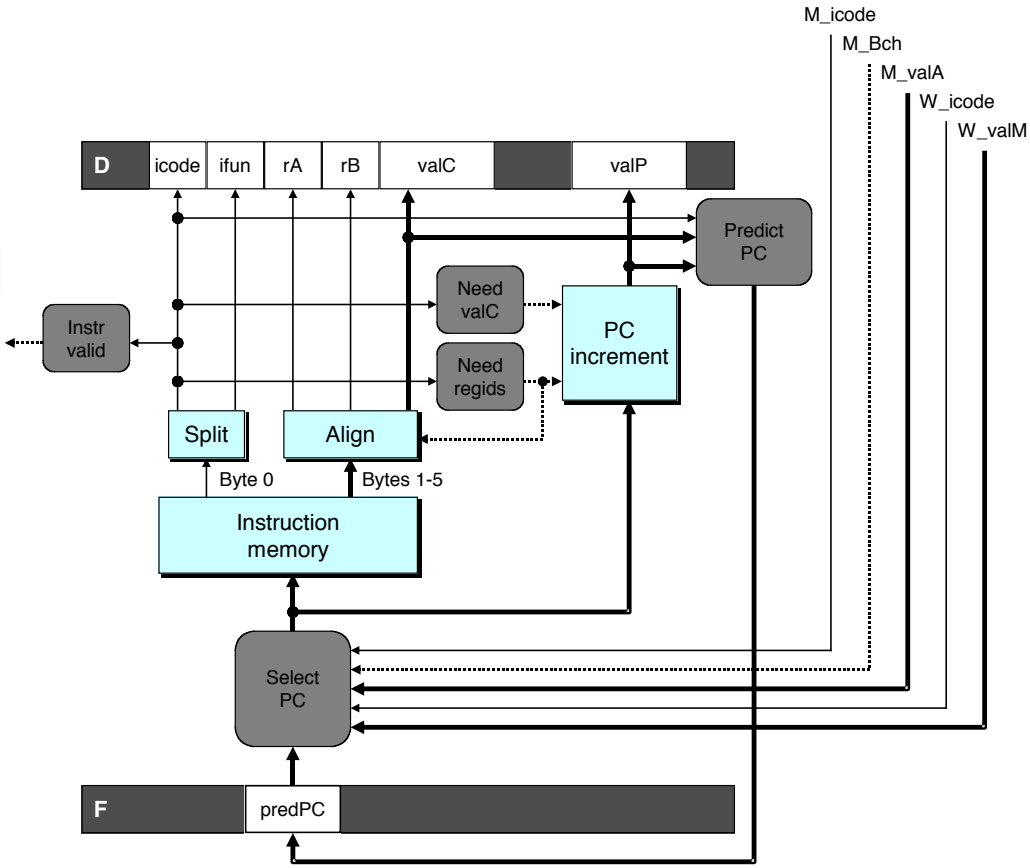
- **e.g.** `jle` **0xffff0a42**
- **Where to jump next depends on results of condition codes of the previous instruction, which has still not even reached the Execute stage where the condition codes are generated!**
- ***Branch prediction strategy*: Predict next PC to be the jump address/destination (valC)**
  - **So Y86 default is to always take the branch – *always taken* branch prediction strategy**
  - **Typically right 60% of time**
- **Begin *speculative execution* - Only correct if branch is taken**
  - **If incorrect, must flush pipeline and discard speculative execution results**

Our PC Prediction Strategy

Return Instruction: `ret`

- Where to jump next depends on popping the return address off the stack, which means pulling it from memory – an operation that occurs many stages in the future!
- Don't try to predict

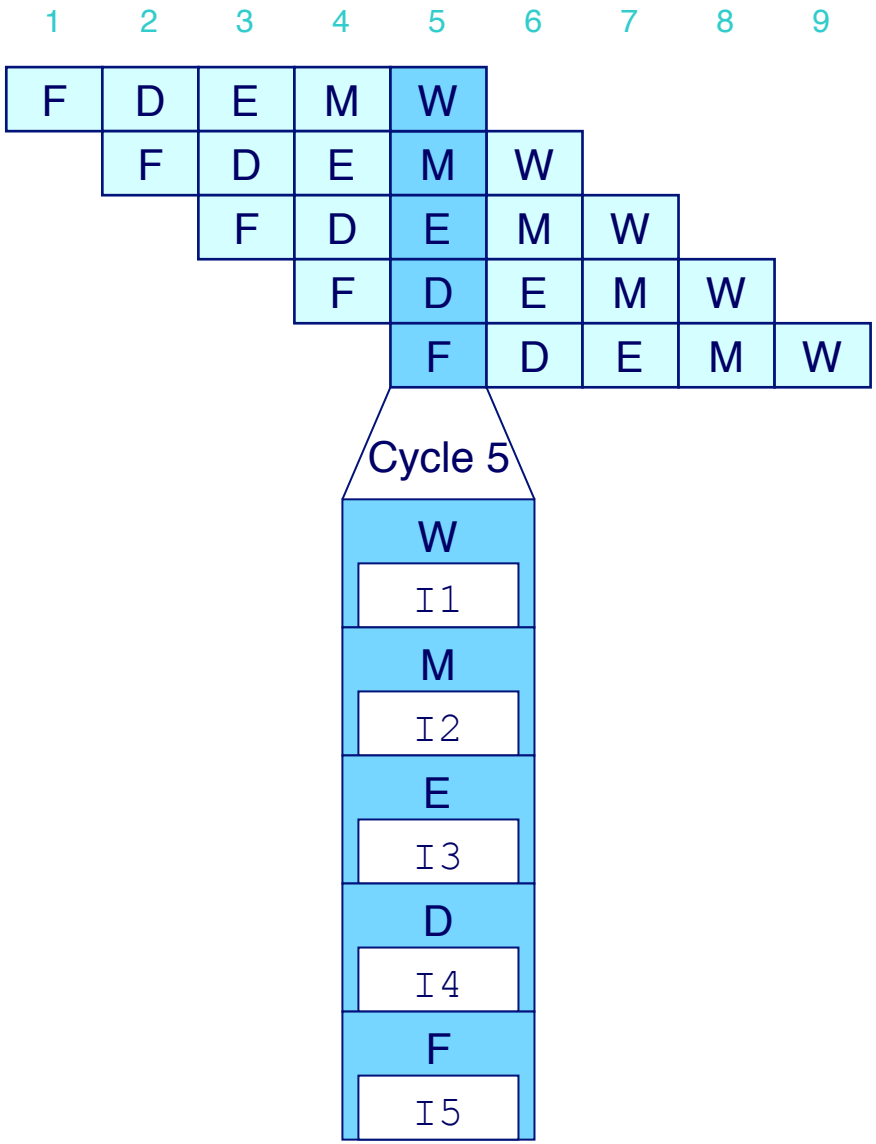
Recovering from PC Misprediction



- **Mispredicted Jump**
  - **Will see branch flag once instruction reaches memory stage**
  - **Can get fall-through PC from valP**
  - **Have to flush/ignore much of pipeline, fetch correct instruction and start to fill pipeline again**
- **Return Instruction**
  - **Will get return PC when `ret` reaches write-back stage**
  - **Pipeline goes mostly empty**

Pipeline Demonstration

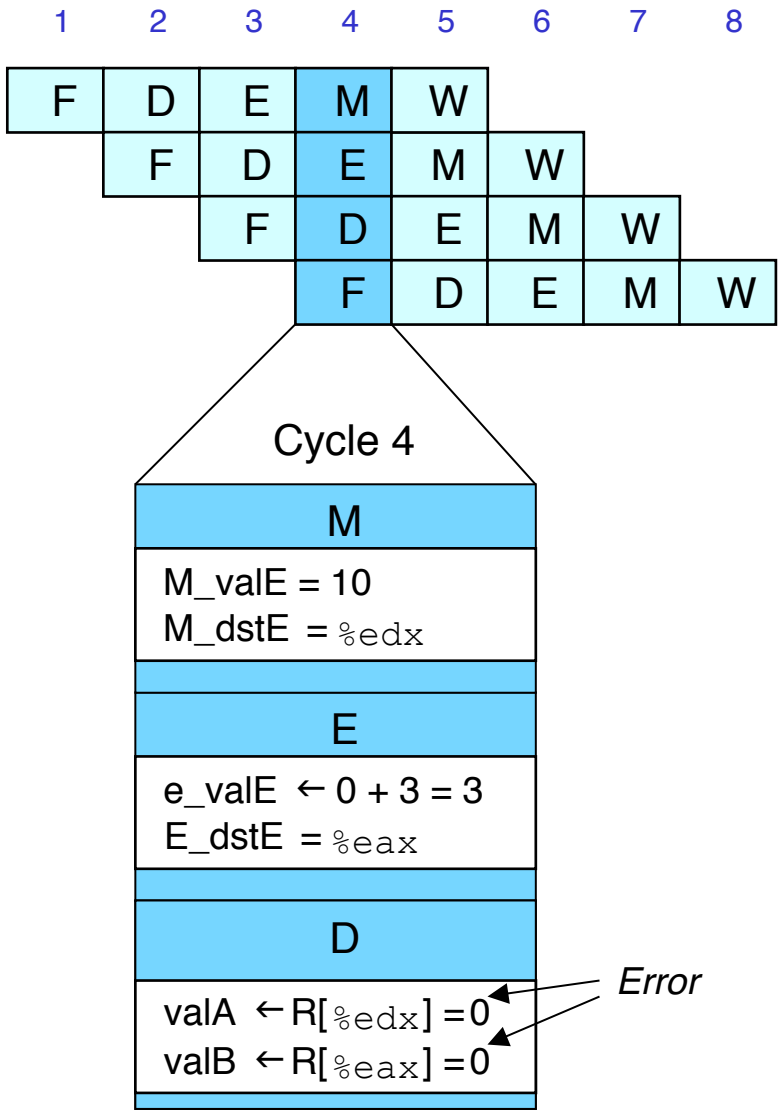
```
irmovl    $1,%eax   #I1
irmovl    $2,%ecx   #I2
irmovl    $3,%edx   #I3
irmovl    $4,%ebx   #I4
halt                #I5
```



File: **`demo-basic.ys`**

Data Dependencies: No Nop

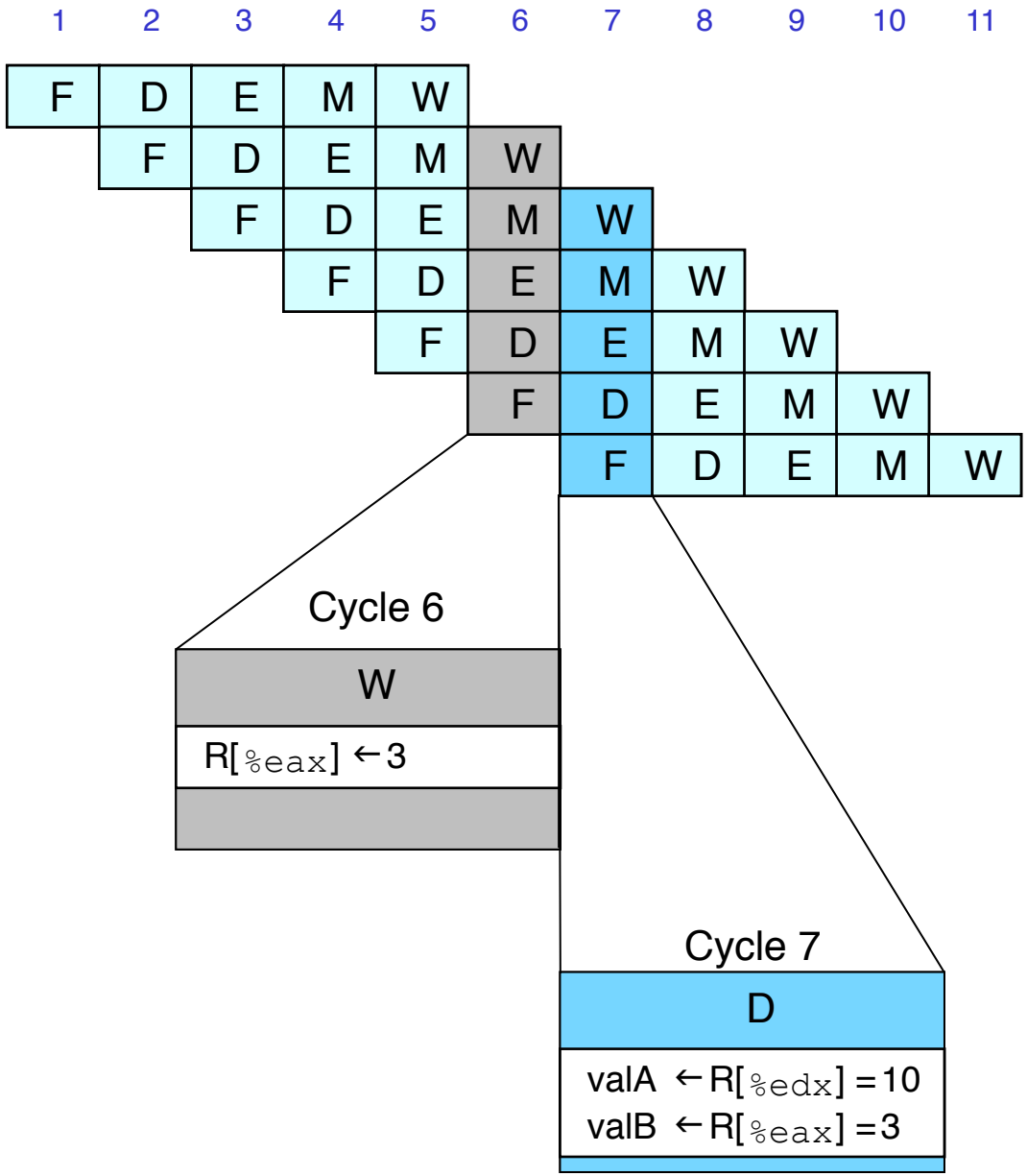
```
# demo-h0.ys
 0x000: irmovl $10,%edx
 0x006: irmovl  $3,%eax
 0x00c: addl %edx,%eax
0x00e: halt
```



Assume all registers initialized to zero

Data Dependencies: 3 Nop's

```
# demo-h3.ys
 0x000: irmovl $10,%edx
 0x006: irmovl  $3,%eax
 0x00c: nop
 0x00d: nop
 0x00e: nop
 0x00f: addl %edx,%eax
0x011: halt
```



Supplementary Slides

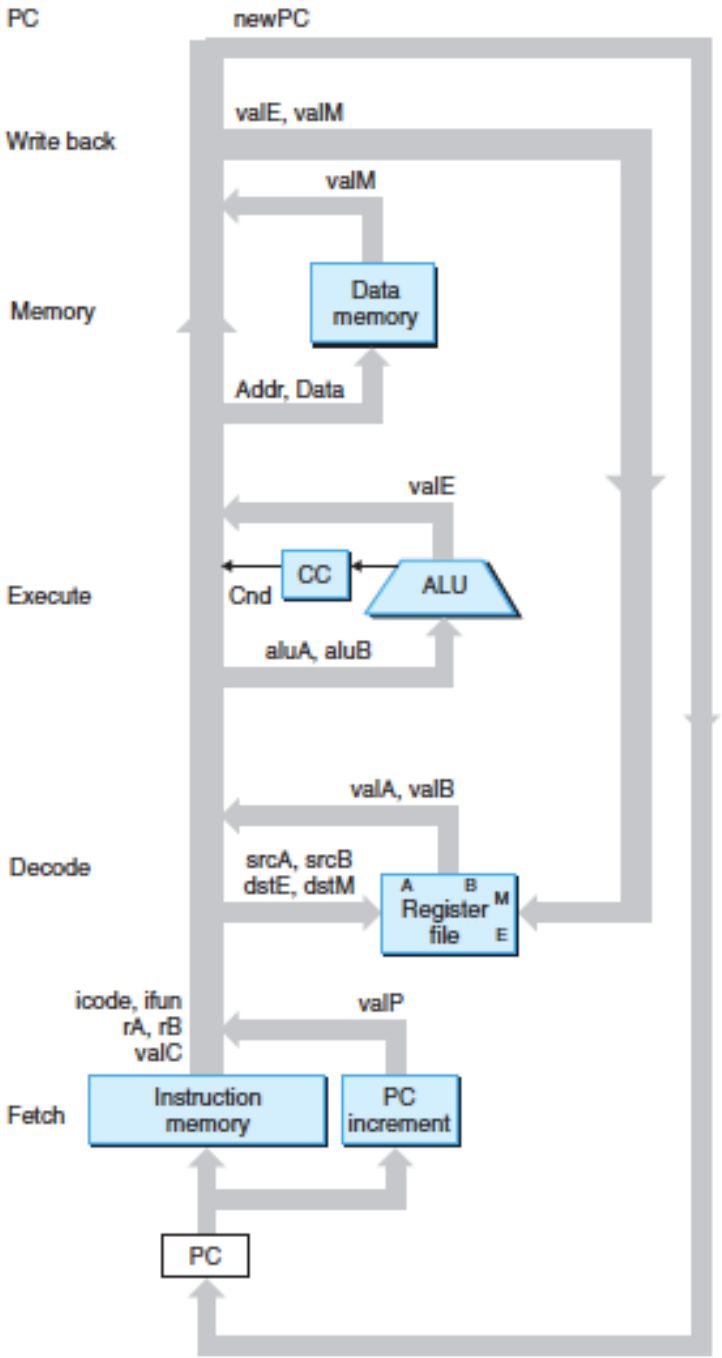
Y86 SEQ Hardware Structure

State

- Program counter register (PC)
- Condition code register (CC)
- Register File
- Memories
  - Access same memory space
  - Data: for reading/writing program data
  - Instruction: for reading instructions

Instruction Flow

- Read instruction at address specified by PC
- Process through stages
- Update program counter



SEQ Stages

Fetch

- Read instruction from instruction memory

Decode

- Read program registers

Execute

- Compute value or address

Memory

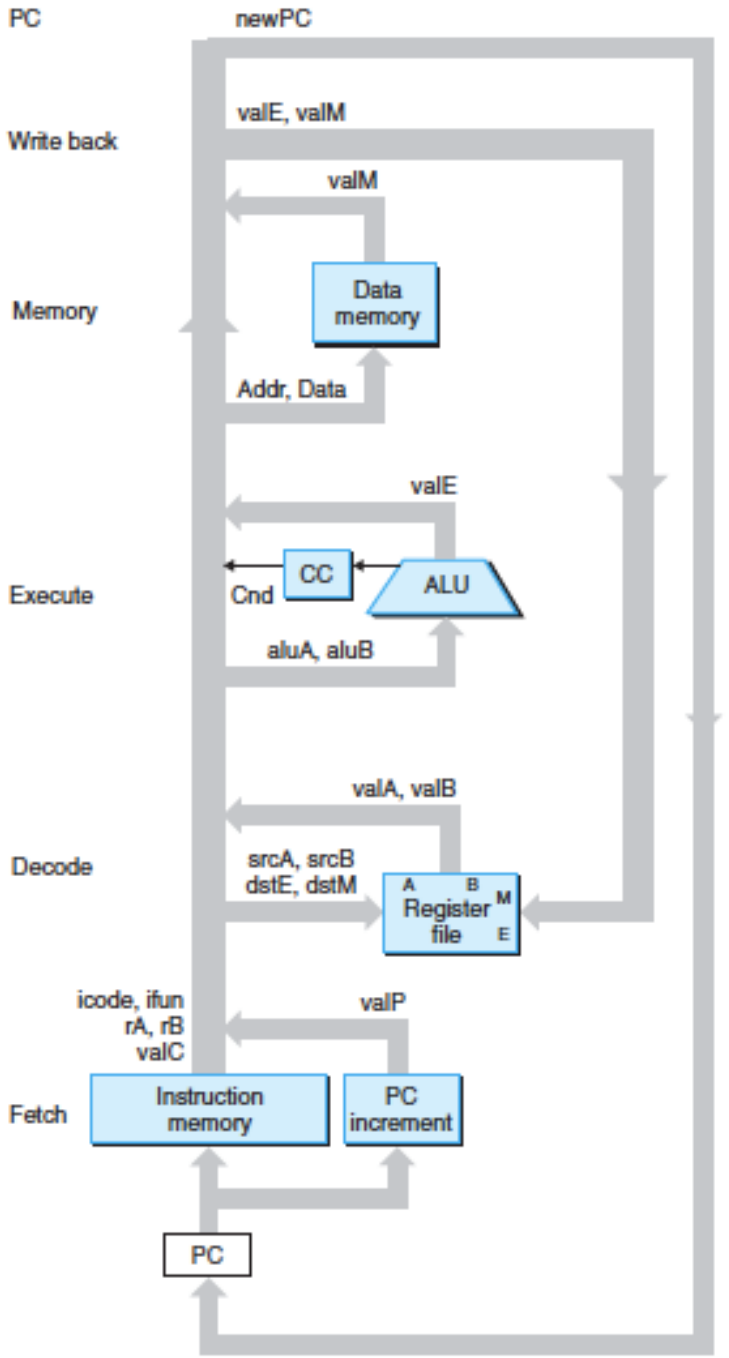
- Read or write data

Write Back

- Write program registers

PC

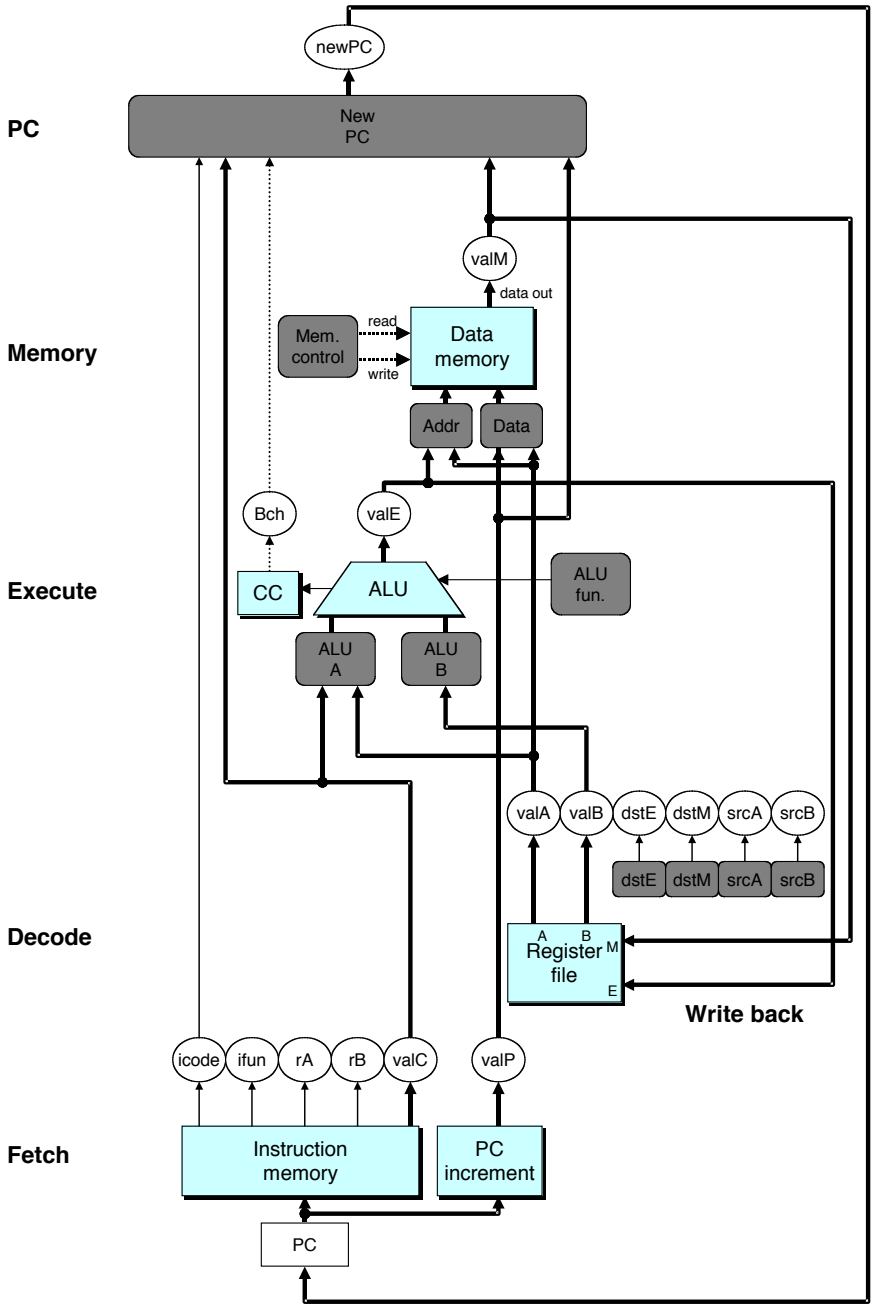
- Update program counter



SEQ Hardware

Key

- Blue boxes: predesigned hardware blocks
  - E.g., memories, ALU
- Gray boxes: control logic
  - Describe in HCL
- White ovals: labels for signals
- Thick lines: 32-bit word values
- Thin lines: 4-8 bit values
- Dotted lines: 1-bit values



Hardware Control Language

- Very simple hardware description language
- Can only express limited aspects of hardware operation
  - Parts we want to explore and modify

Data Types

- `bool`: Boolean
  - `a`, `b`, `c`, ...
- `int`: words
  - `A`, `B`, `C`, ...
  - Does not specify word size---bytes, 32-bit words, ...

Statements

- `bool a =` ***`bool-expr`*** `;`
- `int A =` ***`int-expr`*** `;`

HCL Operations

- Classify by type of value returned

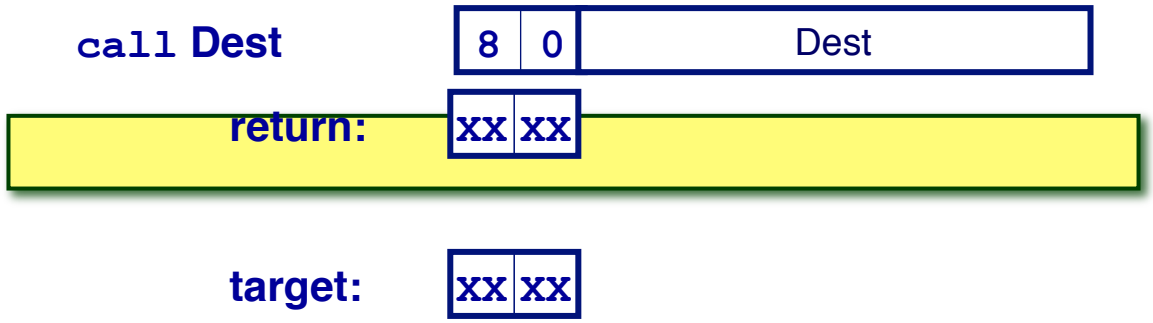
Boolean Expressions

- Logic Operations
  - `a && b, a || b, !a`
- Word Comparisons
  - `A == B, A != B, A < B, A <= B, A >= B, A > B`
- Set Membership
  - `A in { B, C, D }`
    - » Same as `A == B || A == C || A == D`

Word Expressions

- Case expressions
  - `[ a : A; b : B; c : C ]`
  - Evaluate test expressions `a`, `b`, `c`, … in sequence
  - Return word expression `A`, `B`, `C`, … for first successful test

Executing `call`



Fetch

- Read 5 bytes
- Increment PC by 5

Decode

- Read stack pointer

Execute

- Decrement stack pointer by 4

Memory

- Write incremented PC to new value of stack pointer

Write back

- Update stack pointer

PC Update

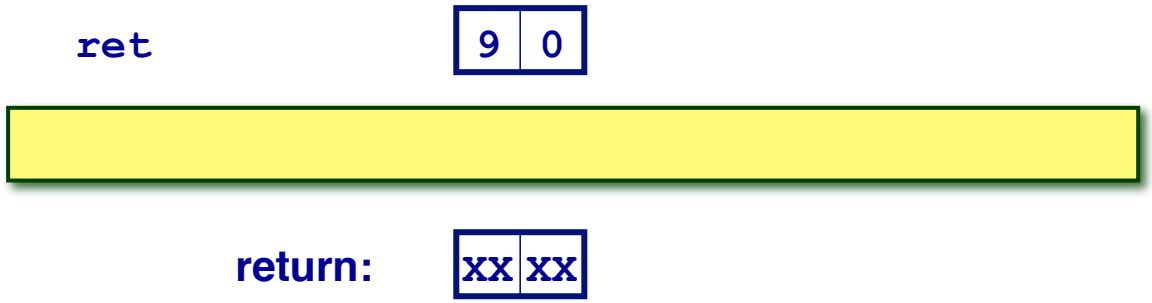
- Set PC to Dest

Stage Computation: `call`

	`call` Dest	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_4[PC+1]	Read destination address
	valP ← PC+5	Compute return point
Decode	valB ← R[`%esp`]	Read stack pointer
Execute	valE ← valB + –4	Decrement stack pointer
Memory	M_4[valE] ← valP	Write return value on stack
Write back	R[`%esp`] ← valE	Update stack pointer
PC update	PC ← valC	Set PC to destination

- Use ALU to decrement stack pointer
- Store incremented PC

Executing `ret`



Fetch

- Read 1 byte

Decode

- Read stack pointer

Execute

- Increment stack pointer by 4

Memory

- Read return address from old stack pointer

Write back

- Update stack pointer

PC Update

- Set PC to return address

Stage Computation: `ret`

	`ret`	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
Decode	valA ← R[%esp] valB ← R[%esp]	Read operand stack pointer Read operand stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M_4[valA]	Read return address
Write back	R[%esp] ← valE	Update stack pointer
PC update	PC ← valM	Set PC to return address

- Use ALU to increment stack pointer
- Read return address from memory

Computation Steps

		OPl rA, rB	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB	rA:rB ← M_1[PC+1]	Read register byte
	valC		[Read constant word]
	valP	valP ← PC+2	Compute next PC
Decode	valA, srcA	valA ← R[rA]	Read operand A
	valB, srcB	valB ← R[rB]	Read operand B
Execute	valE	valE ← valB OP valA	Perform ALU operation
	Cond code	Set CC	Set condition code register
Memory	valM		[Memory read/write]
Write back	dstE	R[rB] ← valE	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	PC ← valP	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

Computation Steps

		`call` Dest	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB		[Read register byte]
	valC	valC ← M_4[PC+1]	Read constant word
	valP	valP ← PC+5	Compute next PC
Decode	valA, srcA		[Read operand A]
	valB, srcB	valB ← R[`%esp`]	Read operand B
Execute	valE	valE ← valB + –4	Perform ALU operation
	Cond code		[Set condition code reg.]
Memory	valM	M_4[valE] ← valP	[Memory read/write]
Write back	dstE	R[`%esp`] ← valE	[Write back ALU result]
	dstM		Write back memory result
PC update	PC	PC ← valC	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

Computed Values

Fetch

icode	Instruction code
ifun	Instruction function
rA	Instr. Register A
rB	Instr. Register B
valC	Instruction constant
valP	Incremented PC

Decode

srcA	Register ID A
srcB	Register ID B
dstE	Destination Register E
dstM	Destination Register M
valA	Register value A
valB	Register value B

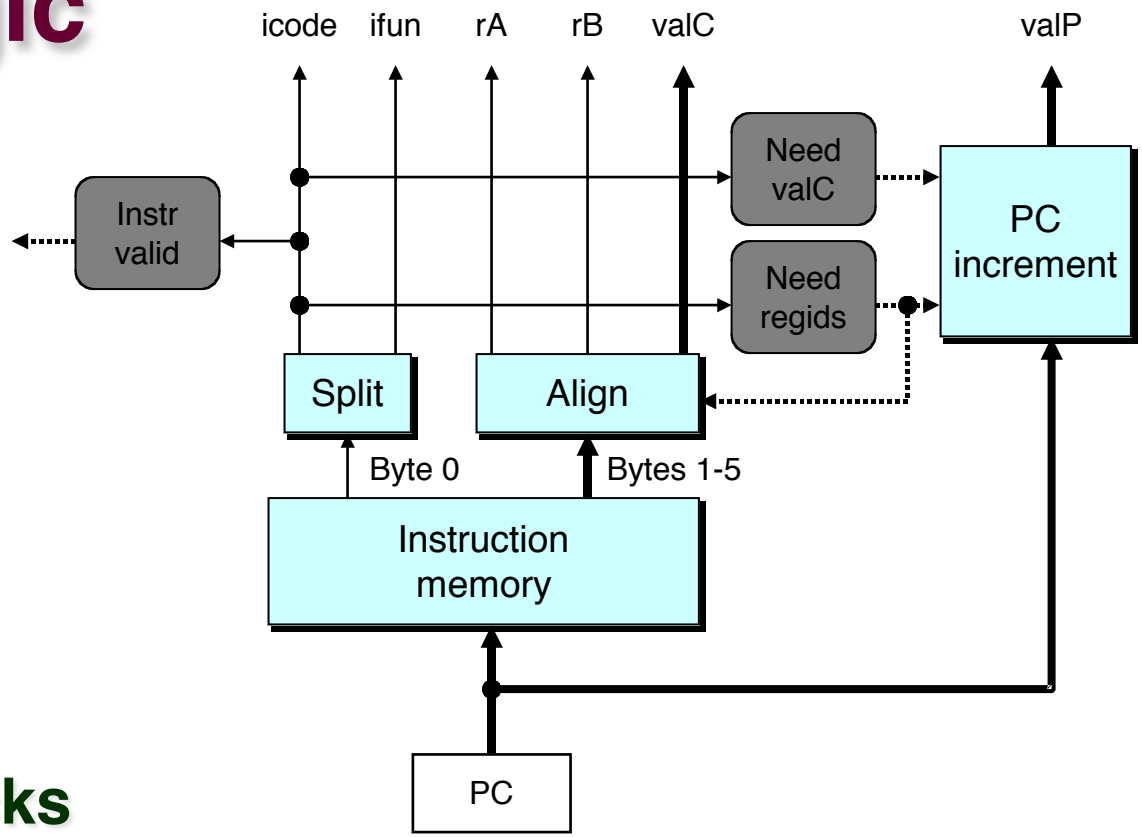
Execute

- valE ALU result
- Bch Branch flag

Memory

- valM Value from memory

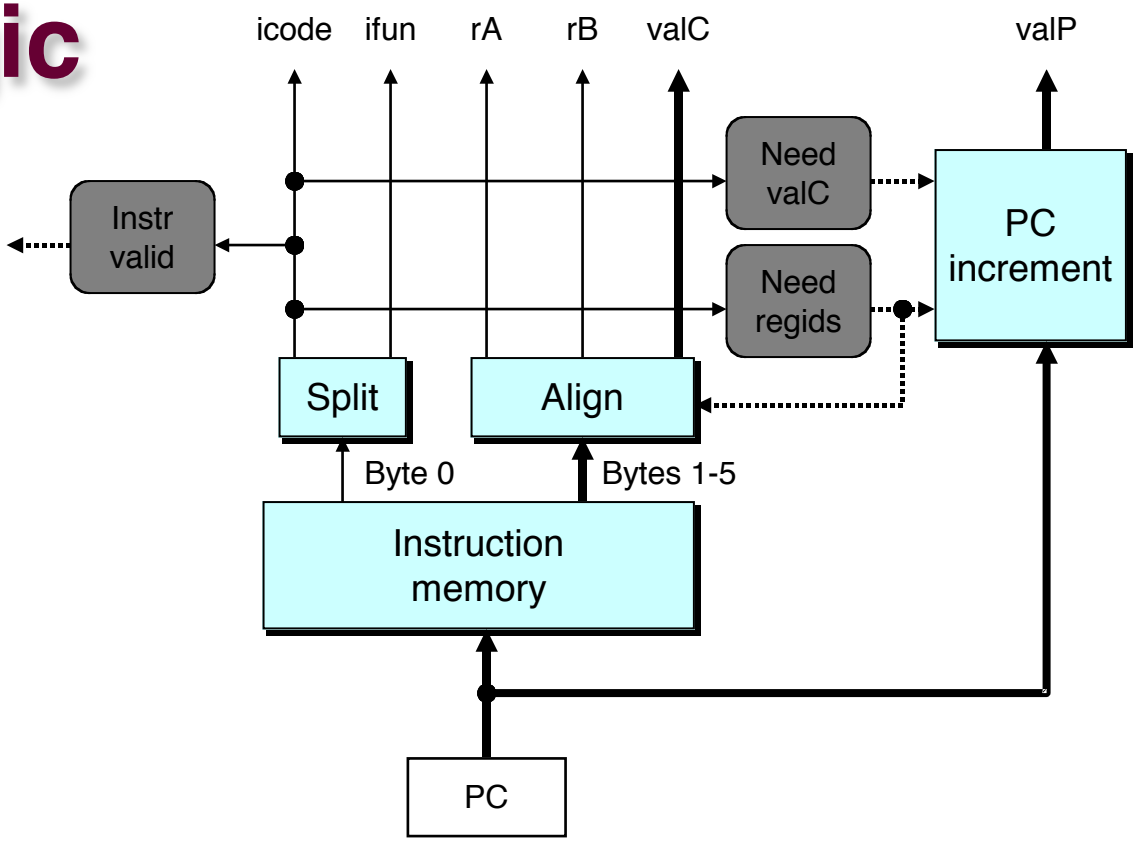
Fetch Logic



Predefined Blocks

- **PC: Register containing PC**
- **Instruction memory: Read 6 bytes (PC to PC+5)**
- **Split: Divide instruction byte into icode and ifun**
- **Align: Get fields for rA, rB, and valC**

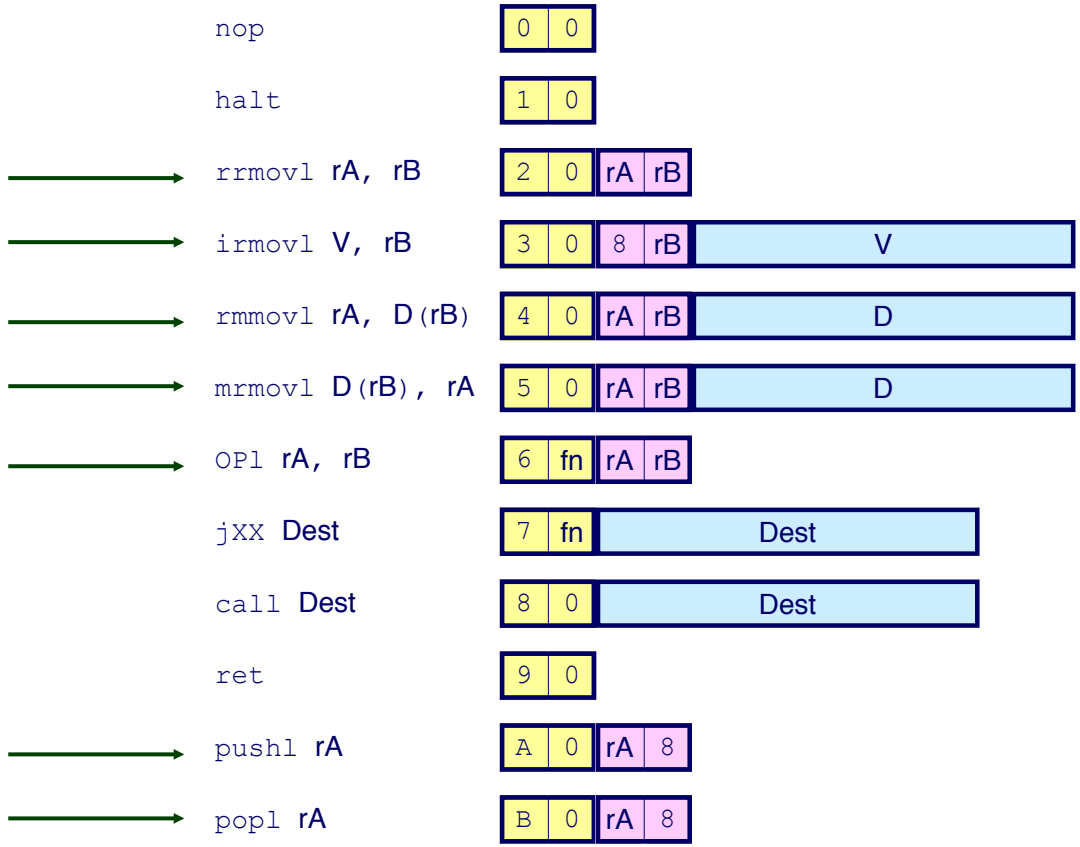
Fetch Logic



Control Logic

- **Instr. Valid: Is this instruction valid?**
- **Need regids: Does this instruction have a register bytes?**
- **Need valC: Does this instruction have a constant word?**

Fetch Control Logic



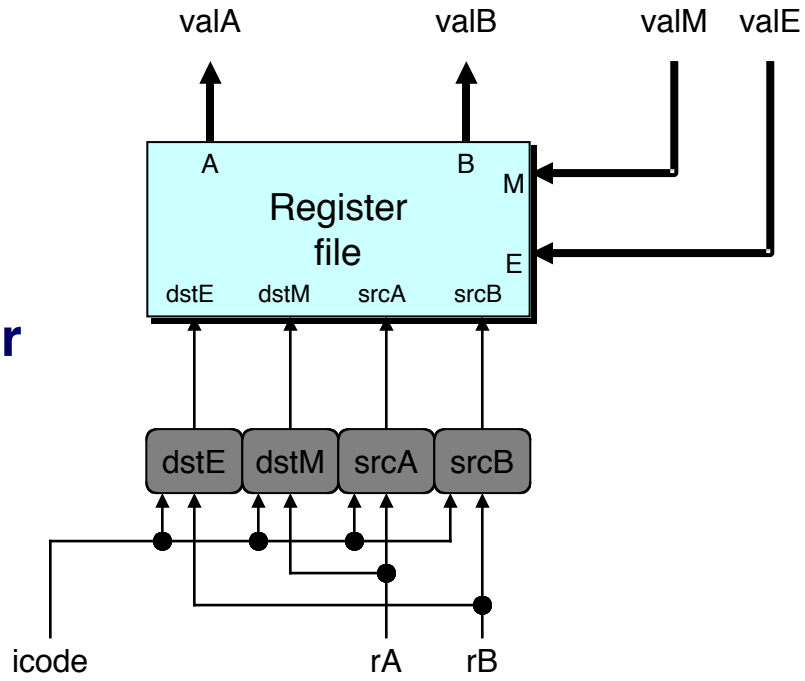

```
bool need_regids =
        icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                   IIRMOVL, IRMMOVL, IMRMOVL };

bool instr_valid = icode in
        { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
          IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
```

Decode Logic

Register File

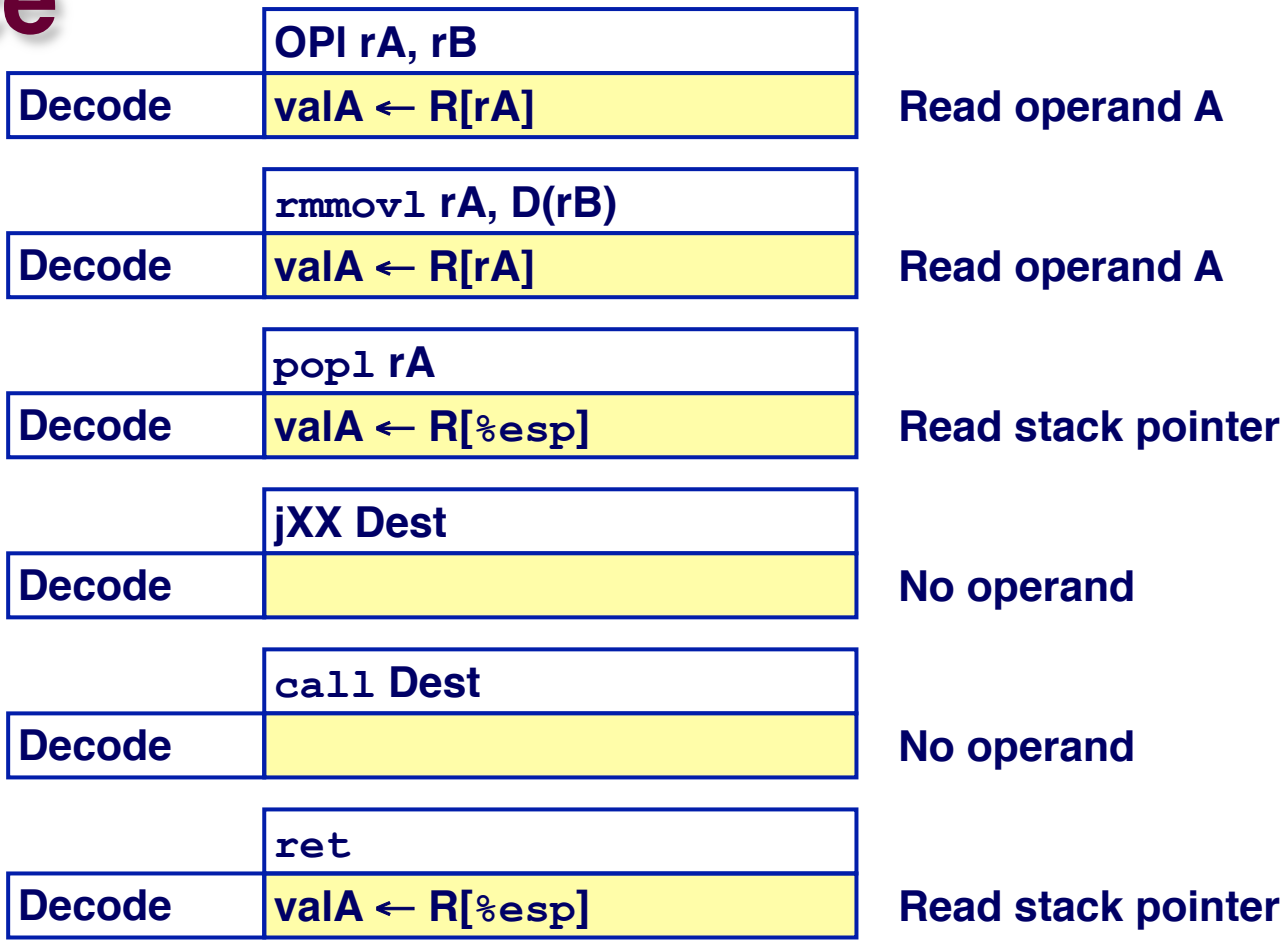
- Read ports A, B
- Write ports E, M
- Addresses are register IDs or 8 (no access)



Control Logic

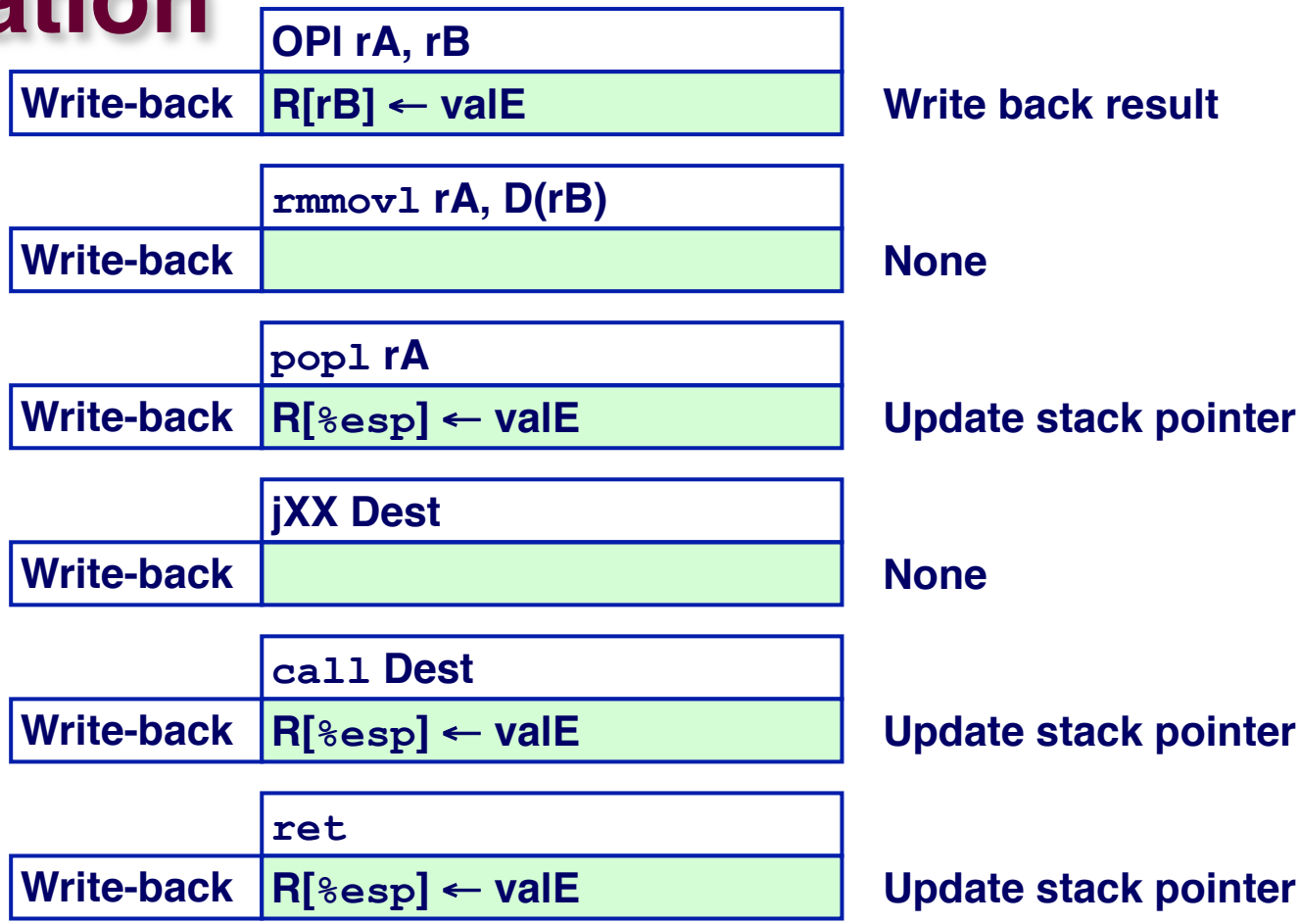
- srcA, srcB: read port addresses
- dstA, dstB: write port addresses

A Source




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

E Destination

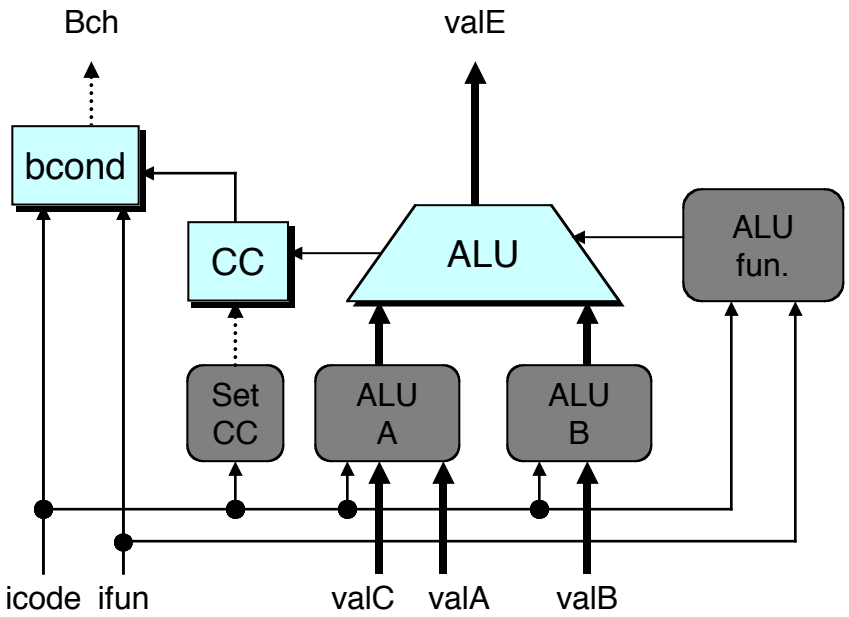



```
int dstE = [
        icode in { IRRMOVL, IIRMOVL, IOPL} : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't need register
];
```

Execute Logic

Units

- ALU
  - Implements 4 required functions
  - Generates condition code values
- CC
  - Register with 3 condition code bits
- bcond
  - Computes branch flag



Control Logic

- Set CC: Should condition code register be loaded?
- ALU A: Input A to ALU
- ALU B: Input B to ALU
- ALU fun: What function should ALU compute?

ALU A Input

	OPl rA, rB	
Execute	valE ← valB OP valA	Perform ALU operation
	`rmmovl` rA, D(rB)	
Execute	valE ← valB + valC	Compute effective address
	`popl` rA	
Execute	valE ← valB + 4	Increment stack pointer
	jXX Dest	
Execute		No operation
	`call` Dest	
Execute	valE ← valB + –4	Decrement stack pointer
	`ret`	
Execute	valE ← valB + 4	Increment stack pointer

```
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
];
```

ALU Operation

	OPl rA, rB	
Execute	valE ← valB OP valA	Perform ALU operation

	`rmmovl` rA, D(rB)	
Execute	valE ← valB + valC	Compute effective address

	`popl` rA	
Execute	valE ← valB + 4	Increment stack pointer

	jXX Dest	
Execute		No operation

	`call` Dest	
Execute	valE ← valB + –4	Decrement stack pointer

	`ret`	
Execute	valE ← valB + 4	Increment stack pointer

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

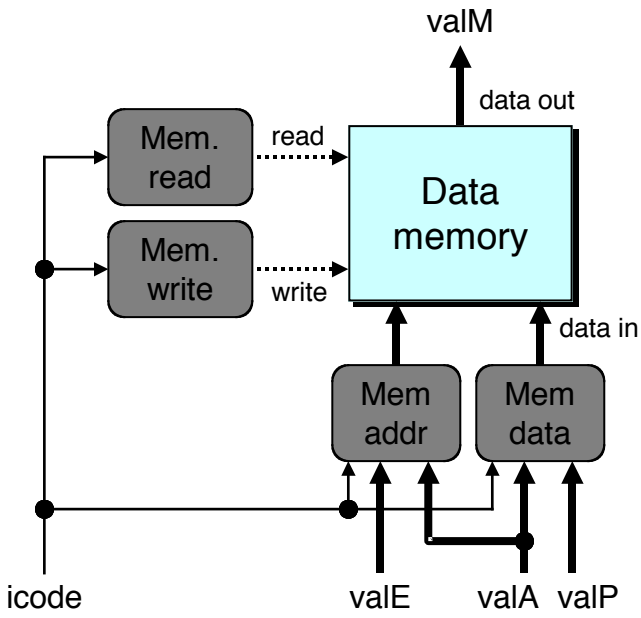
Memory Logic

Memory

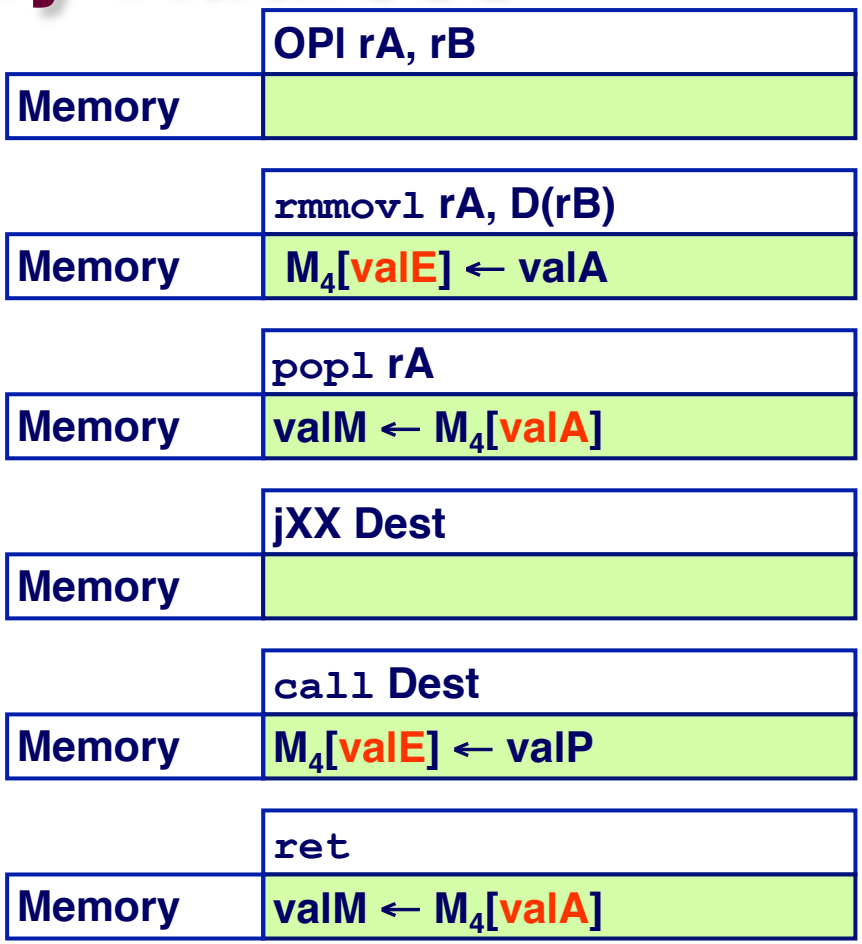
- Reads or writes memory word

Control Logic

- Mem. read: should word be read?
- Mem. write: should word be written?
- Mem. addr.: Select address
- Mem. data.: Select data

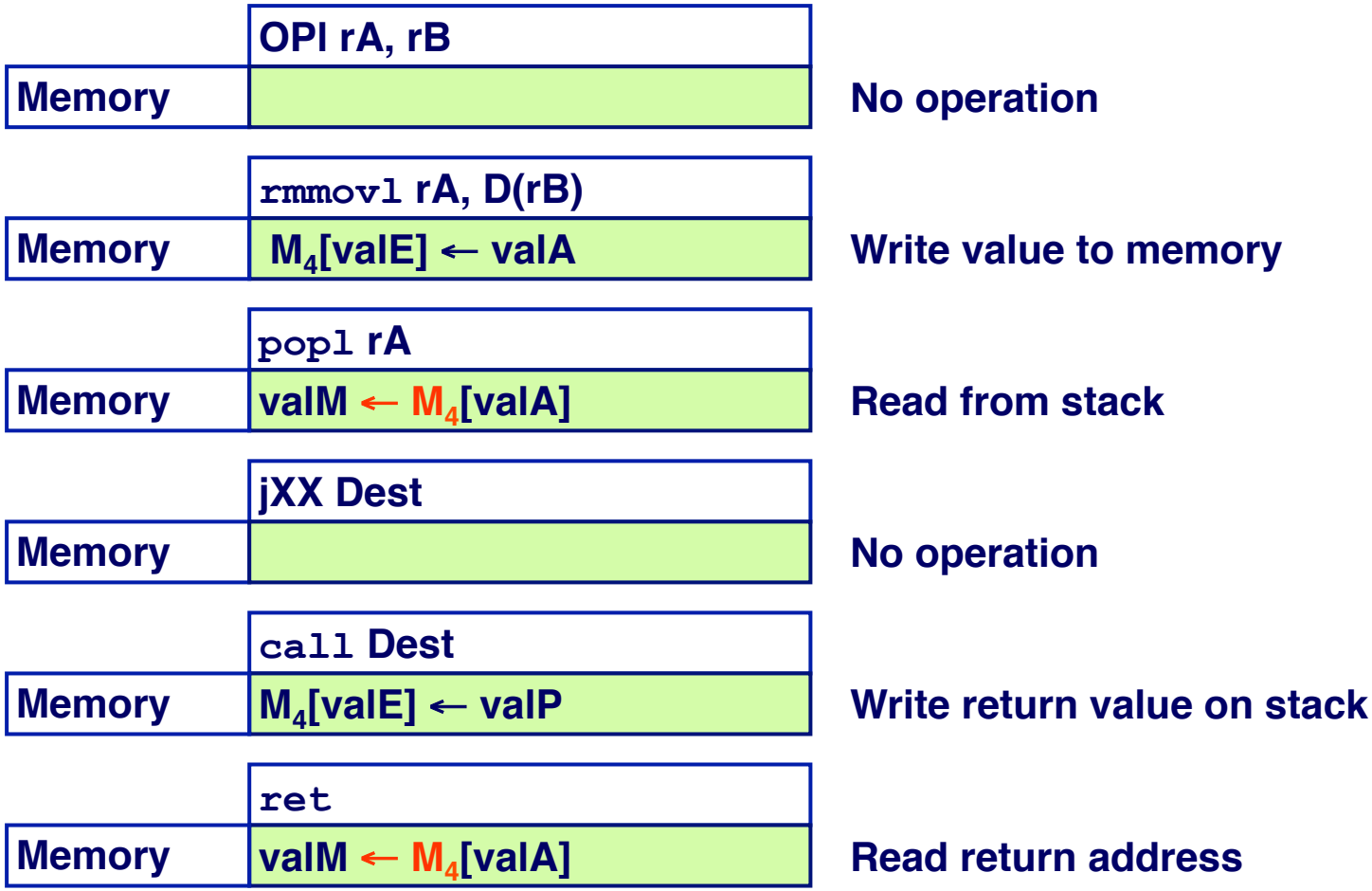


Memory Address




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

Memory Read

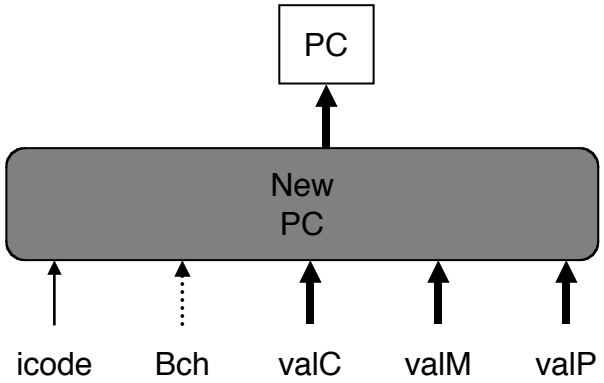



```
bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
```

PC Update Logic

New PC

- Select next value of PC



PC Update

	OPl rA, rB	
PC update	PC ← valP	Update PC

	`rmmovl` rA, D(rB)	
PC update	PC ← valP	Update PC

	`popl` rA	
PC update	PC ← valP	Update PC

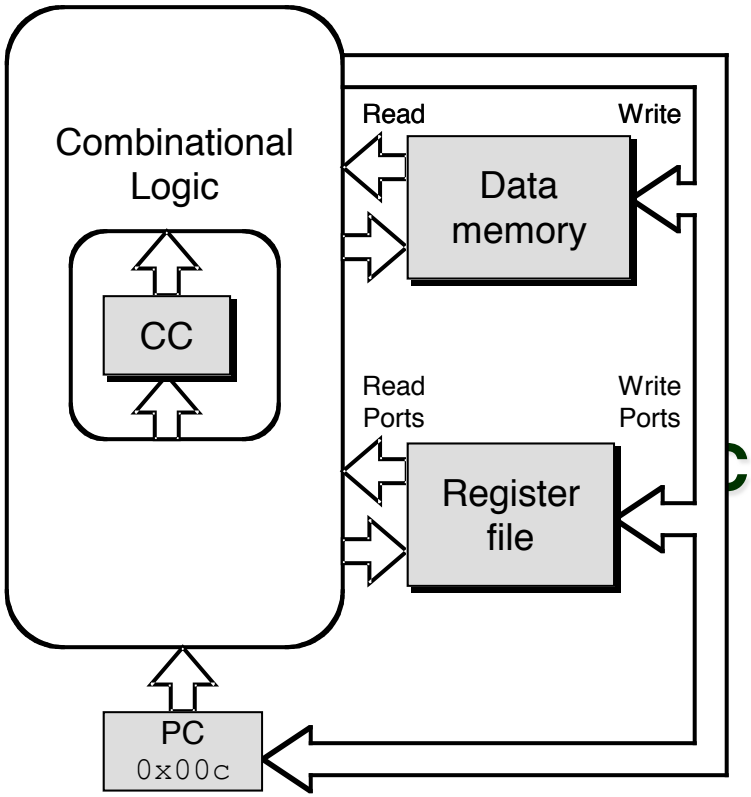
	jXX Dest	
PC update	PC ← Bch ? valC : valP	Update PC

	`call` Dest	
PC update	PC ← valC	Set PC to destination

	`ret`	
PC update	PC ← valM	Set PC to return address

```
int new_pc = [
        icode == ICALL : valC;
        icode == IJXX && Bch : valC;
        icode == IRET : valM;
        1 : valP;
];
```

SEQ Operation



State

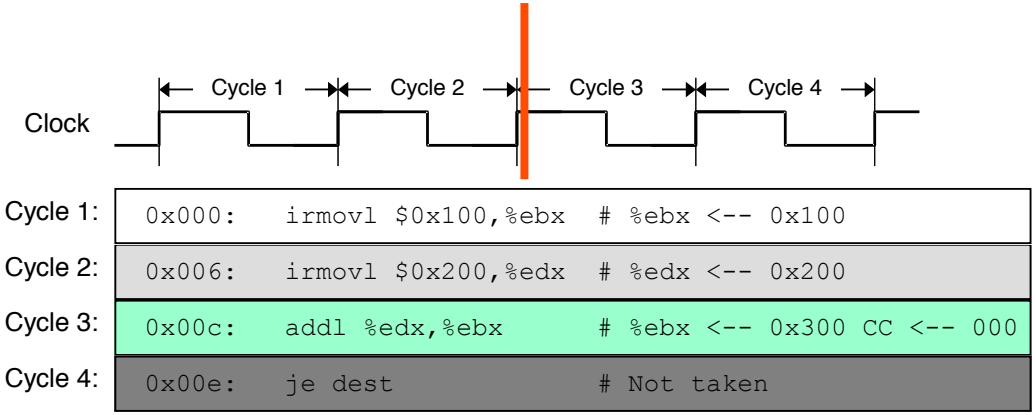
- PC register
- Cond. Code register
- Data memory
- Register file

All updated as clock rises

Combinational Logic

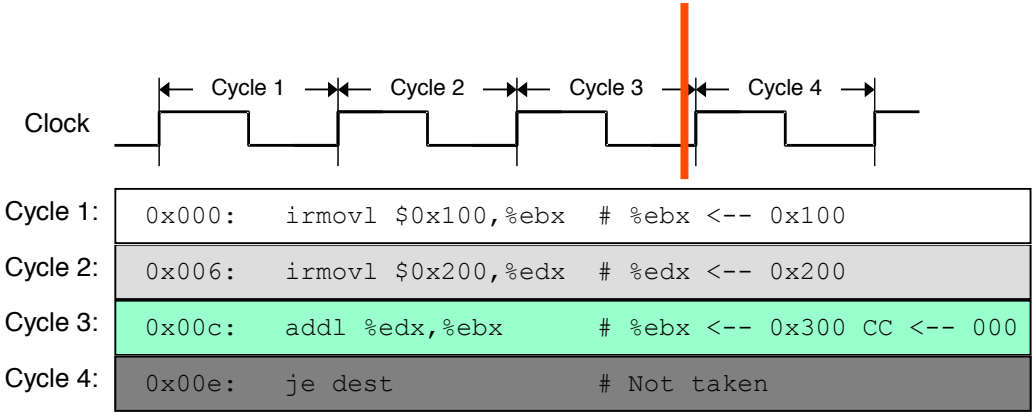
- ALU
- Control logic
- Memory reads
  - Instruction memory
  - Register file
  - Data memory

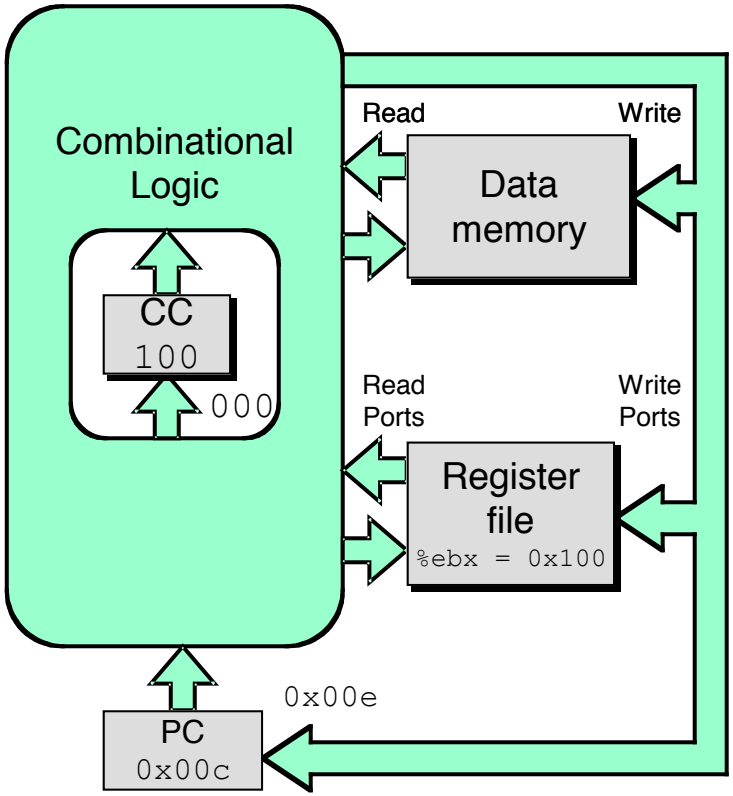
SEQ Operation #2



- state set according to second `irmovl` instruction
- combinational logic starting to react to state changes

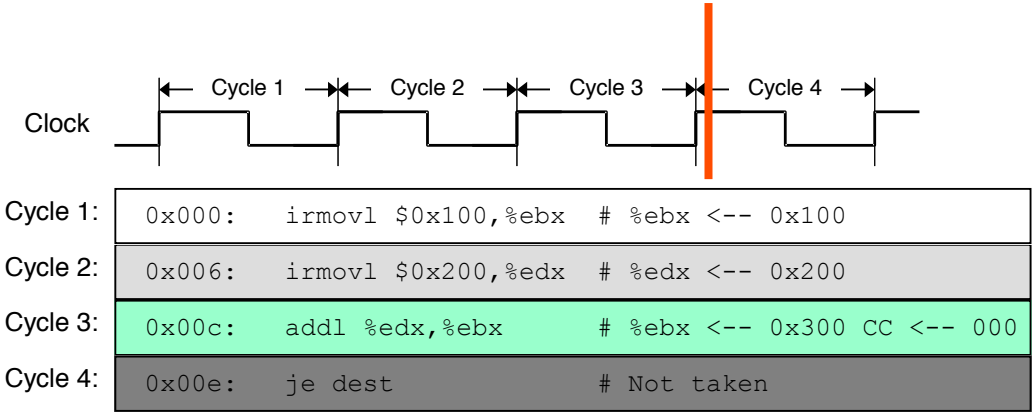
SEQ Operation #3

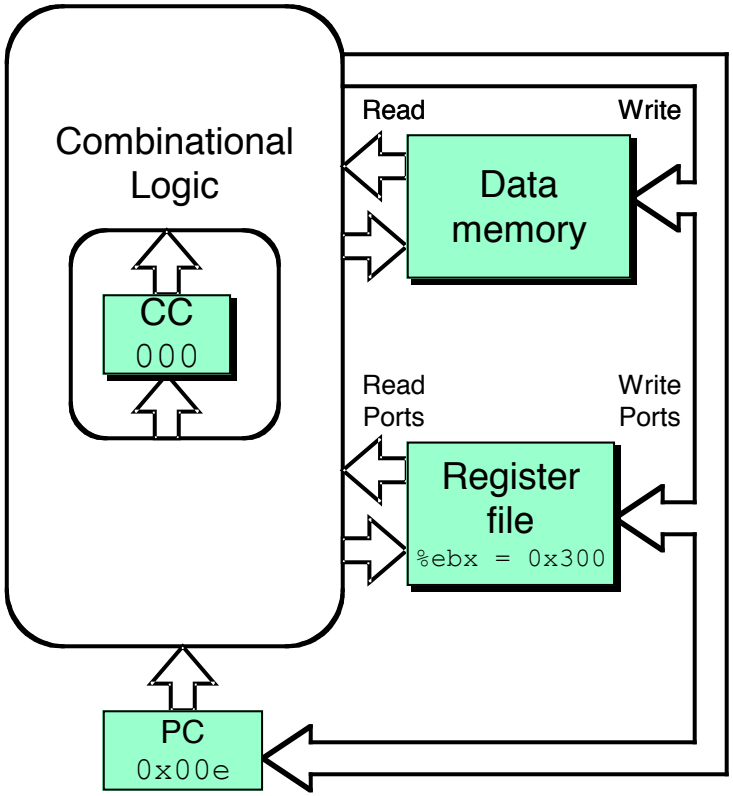




- **state set according to second `irmovl` instruction**
- **combinational logic generates results for `addl` instruction**

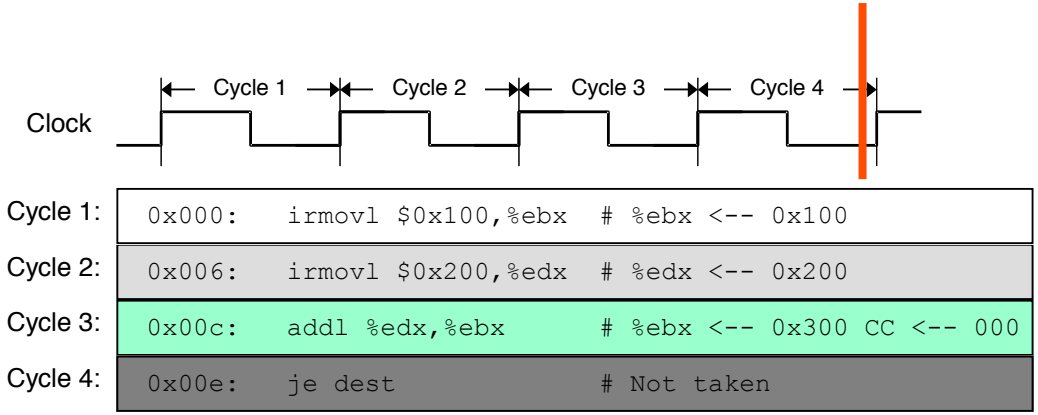
SEQ Operation #4

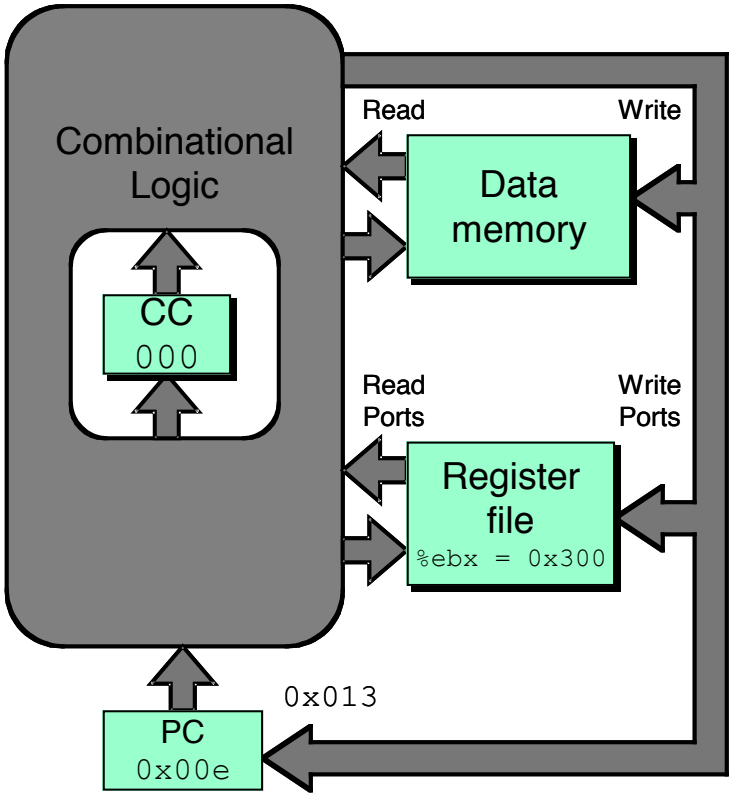




- **state set according to `addl` instruction**
- **combinational logic starting to react to state changes**

SEQ Operation #5





- **state set according to `addl` instruction**
- **combinational logic generates results for `je` instruction**

SEQ Summary

Implementation

- Express every instruction as series of simple steps
- Follow same general flow for each instruction type
- Assemble registers, memories, predesigned combinational blocks
- Connect with control logic

Limitations

- Too slow to be practical
- In one cycle, must propagate through instruction memory, register file, ALU, and data memory
- Would need to run clock very slowly
- Hardware units only active for fraction of clock cycle

SEQ+ Hardware

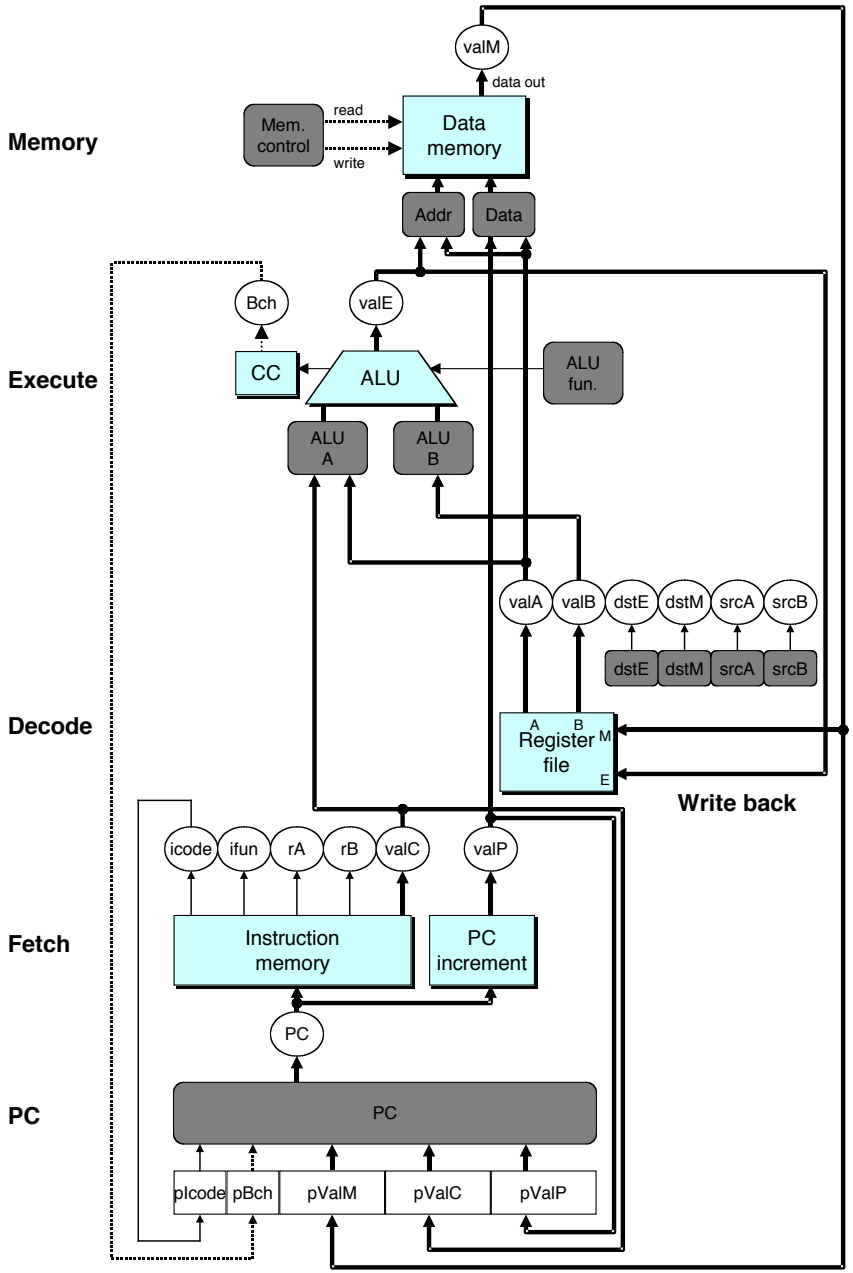
- Still sequential implementation
- Reorder PC stage to put at beginning

PC Stage

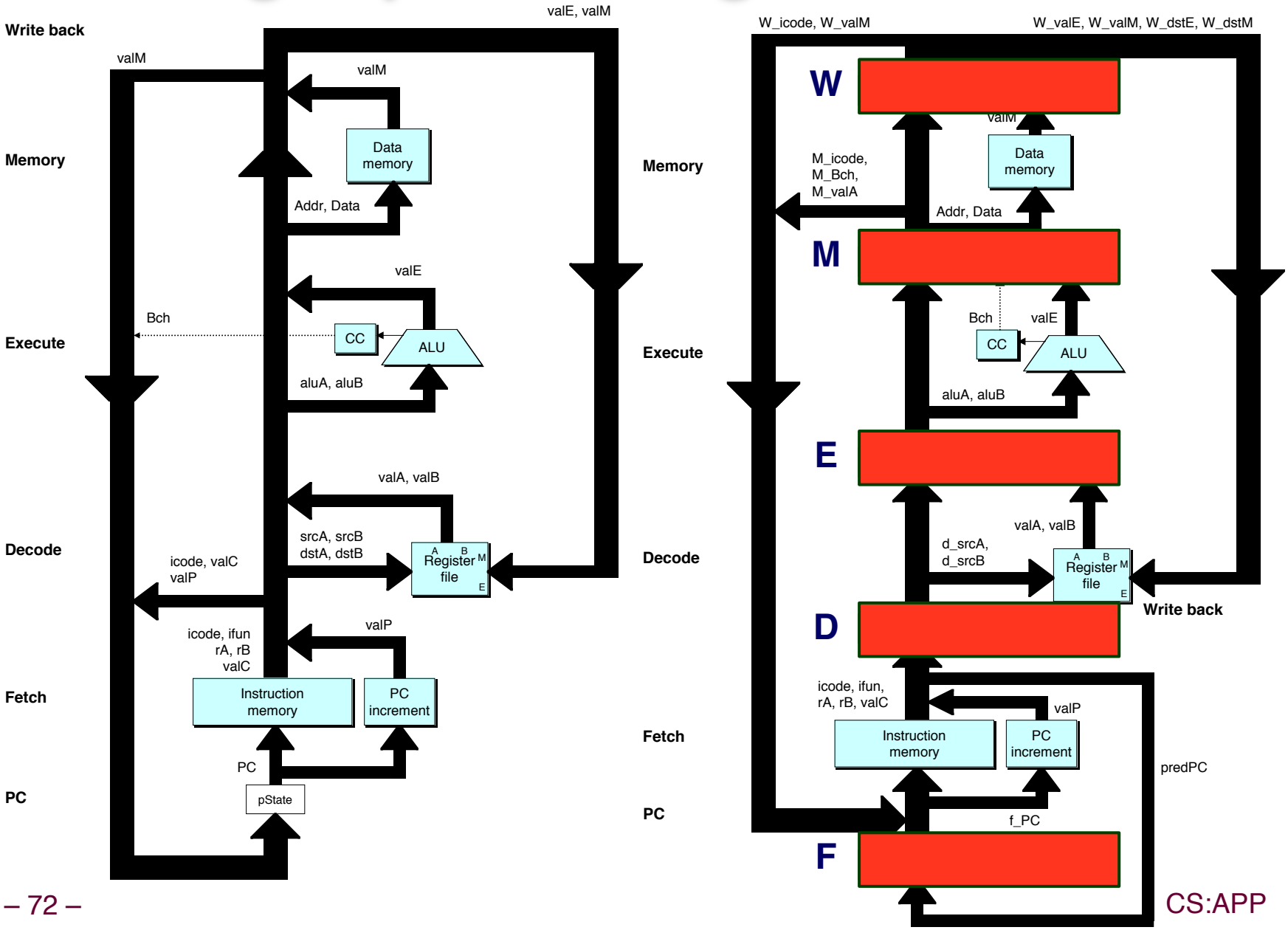
- Task is to select PC for current instruction
- Based on results computed by previous instruction

Processor State

- PC is no longer stored in register
- But, can determine PC based on other stored information



Adding Pipeline Registers



Pipeline Stages

Fetch

- Select current PC
- Read instruction
- Compute incremented PC

Decode

- Read program registers

Execute

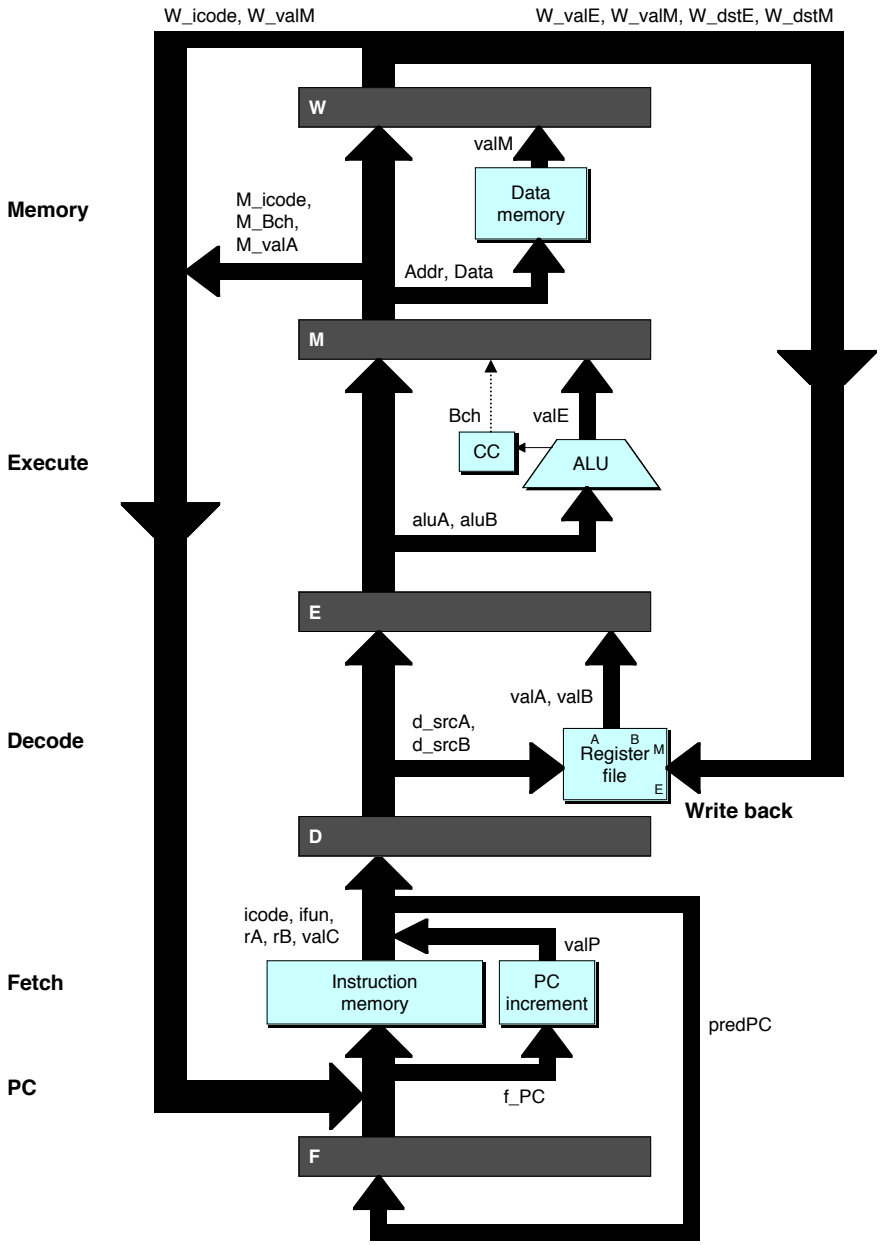
- Operate ALU

Memory

- Read or write data memory

Write Back

- Update register file

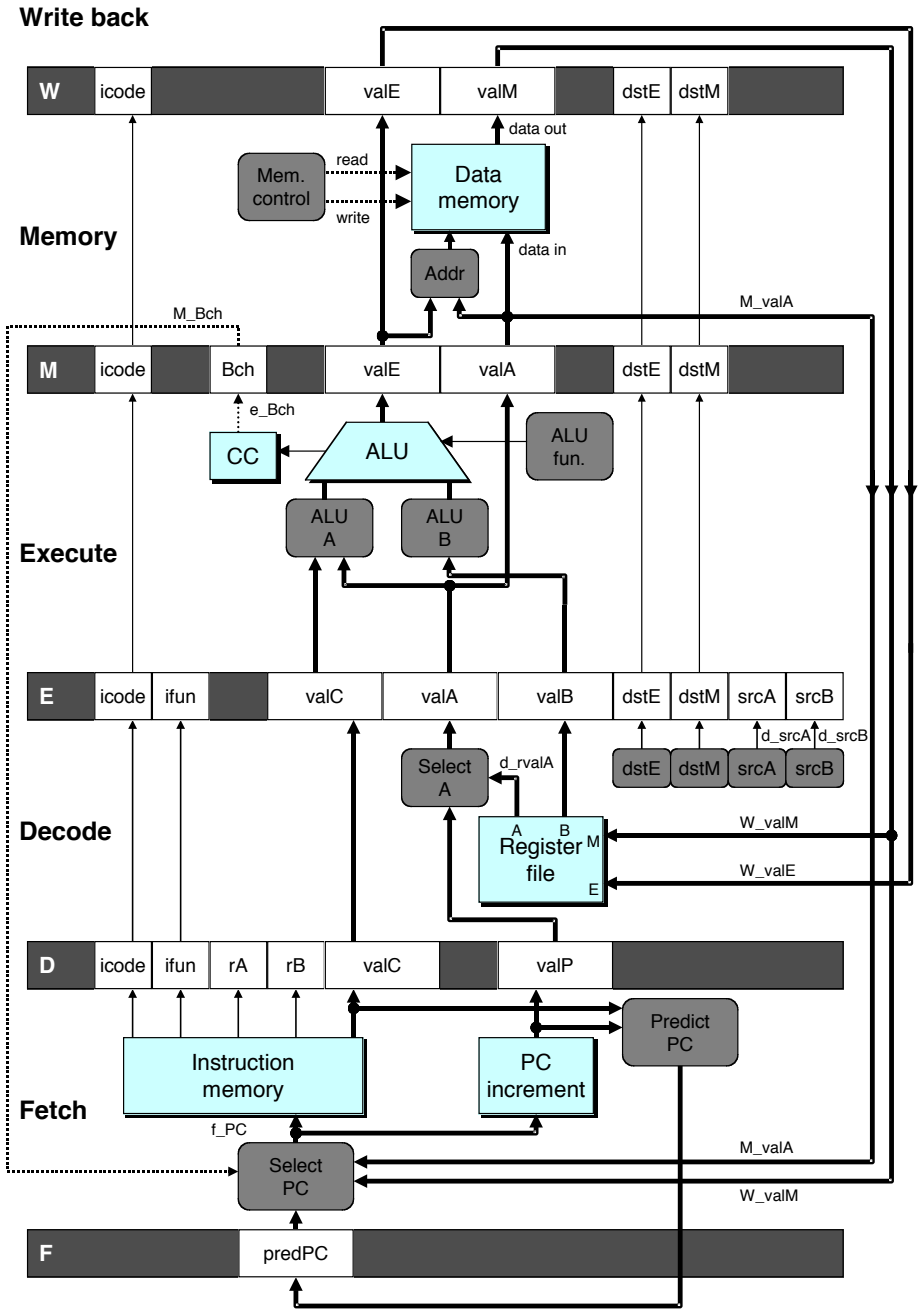


PIPE- Hardware

- Pipeline registers hold intermediate values from instruction execution

Forward (Upward) Paths

- Values passed from one stage to next
- Cannot jump past stages
  - e.g., valC passes through decode



Feedback Paths

Predicted PC

- Guess value of next PC

Branch information

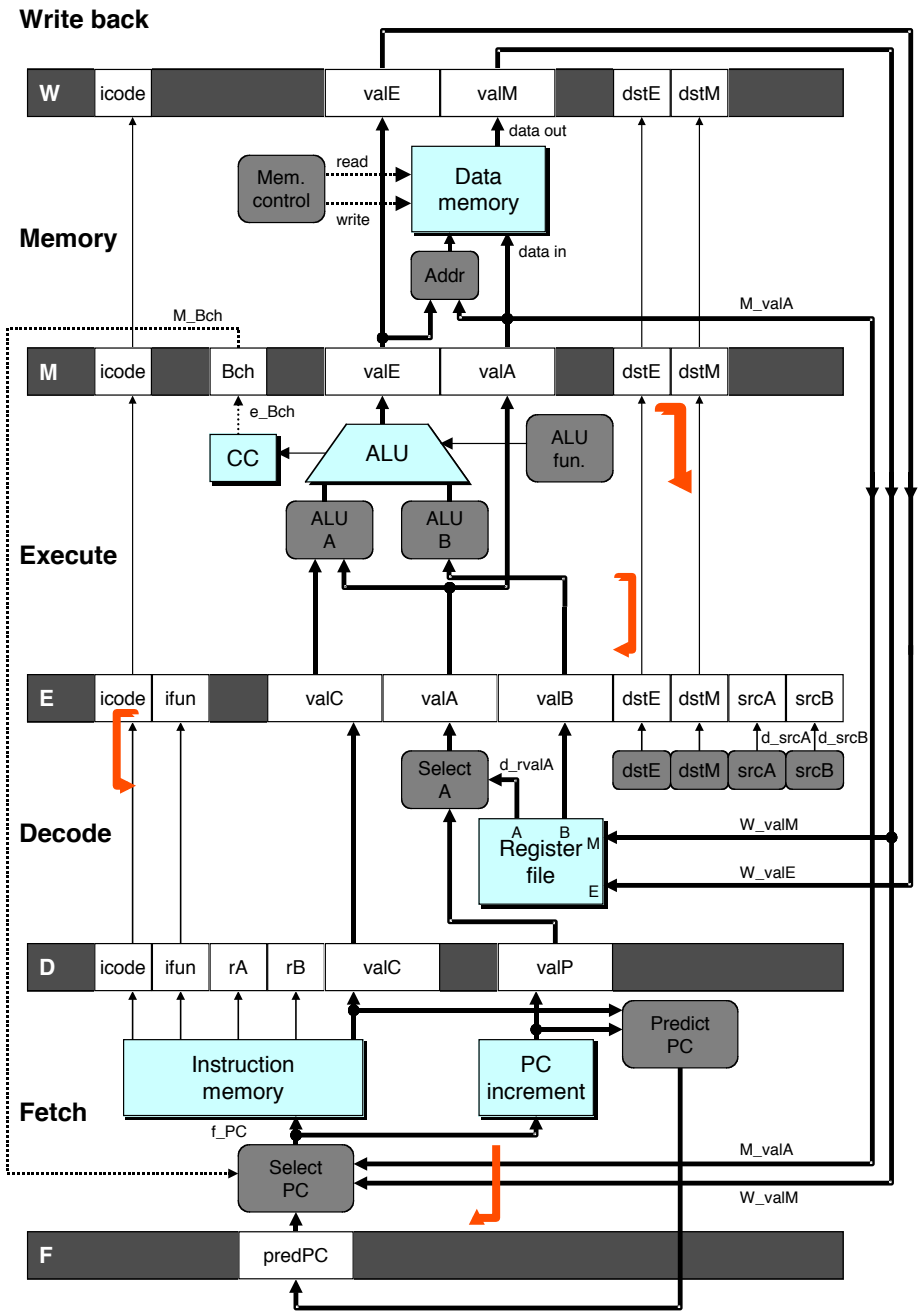
- Jump taken/not-taken
- Fall-through or target address

Return point

- Read from memory

Register updates

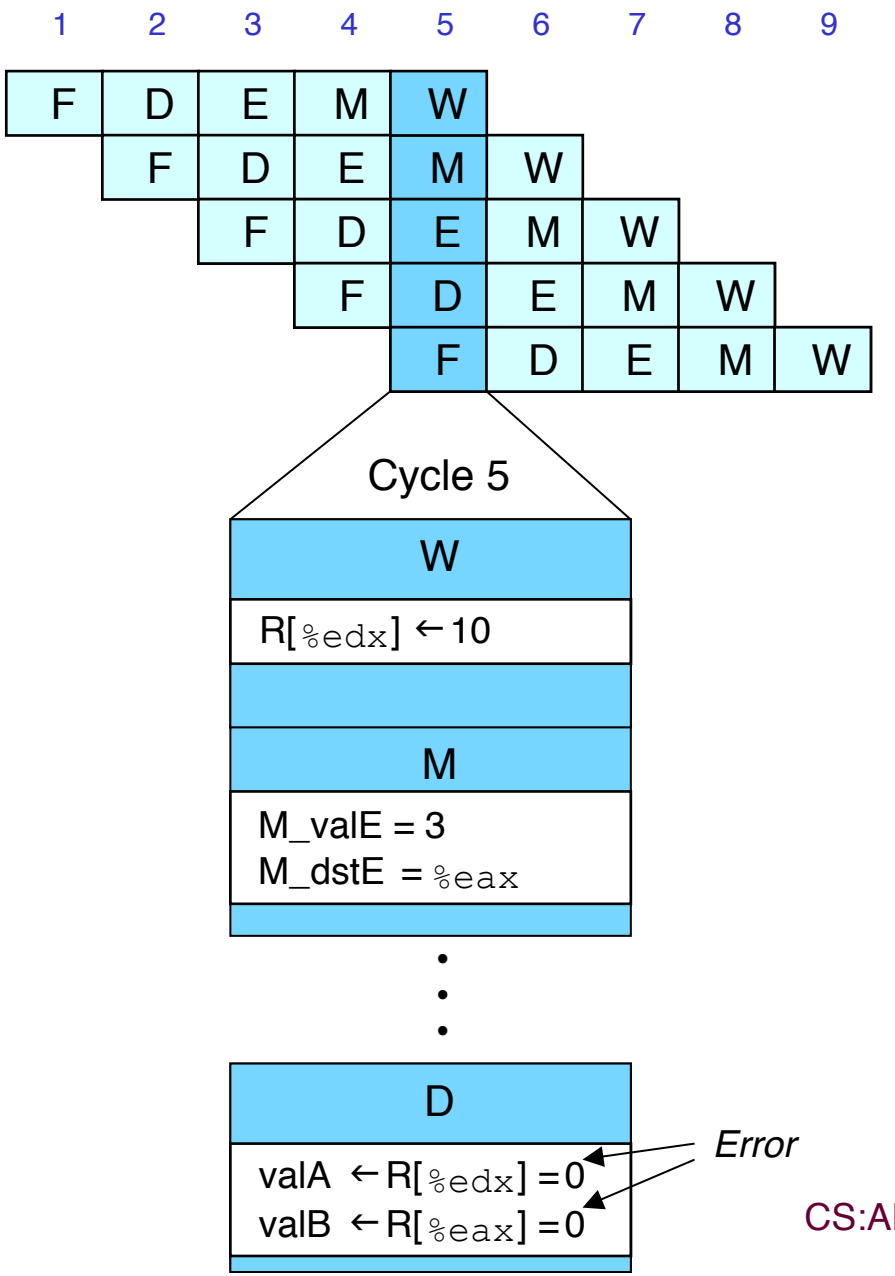
- To register file write ports



Data Dependencies: 1 Nop

```
# demo-h1.ys

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



Data Dependencies: 2 Nop's

```
# demo-h2.ys
 0x000: irmovl $10,%edx
 0x006: irmovl  $3,%eax
 0x00c: nop
 0x00d: nop
 0x00e: addl %edx,%eax
0x010: halt
```