

Sequential Y86-64 Implementations

Lecture 10

October 26th, 2017

Jae W. Lee (jaewlee@snu.ac.kr)

Computer Science and Engineering

Seoul National University

Slide credits: [CS:APP3e] slides from CMU; [COD5e] slides from Elsevier Inc.

Today

Textbook: [CS:APP3e] 4.3

- **SEQ Stage Computation**
- **SEQ Hardware**
- **SEQ Operations**

Recall: Y86-64 Instruction Set #1

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

Recall: Y86-64 Instruction Set #2

Byte	0	1	2	3	4	5	6	
halt	0	0						
nop	1	0						
cmovXX rA, rB	2	fn	rA	rB				rrmovq 2 0
irmovq V, rB	3	0	F	rB	V			cmovle 2 1
rmmovq rA, D(rB)	4	0	rA	rB	D			cmovl 2 2
rrmovq D(rB), rA	5	0	rA	rB	D			cmove 2 3
OPq rA, rB	6	fn	rA	rB				cmovne 2 4
jXX Dest	7	fn	Dest					cmovge 2 5
call Dest	8	0	Dest					cmovg 2 6
ret	9	0						
pushq rA	A	0	rA	F				
popq rA	B	0	rA	F				

Recall: Y86-64 Instruction Set #3

Byte	0	1	2	3	4	5	6	7	8	9				
halt	0	0												
nop	1	0												
cmovXX rA, rB	2	fn	rA	rB										
irmovq V, rB	3	0	F	rB	V									
rmmovq rA, D(rB)	4	0	rA	rB	D									
mrmmovq D(rB), rA	5	0	rA	rB	D									
OPq rA, rB	6	fn	rA	rB	<div><div>addq</div><div>subq</div><div>andq</div><div>xorq</div></div> <div><div>6</div><div>0</div><div>1</div><div>2</div><div>3</div></div>									
jXX Dest	7	fn	Dest											
call Dest	8	0	Dest											
ret	9	0												
pushq rA	A	0	rA	F										
popq rA	B	0	rA	F										

Recall: Y86-64 Instruction Set #4

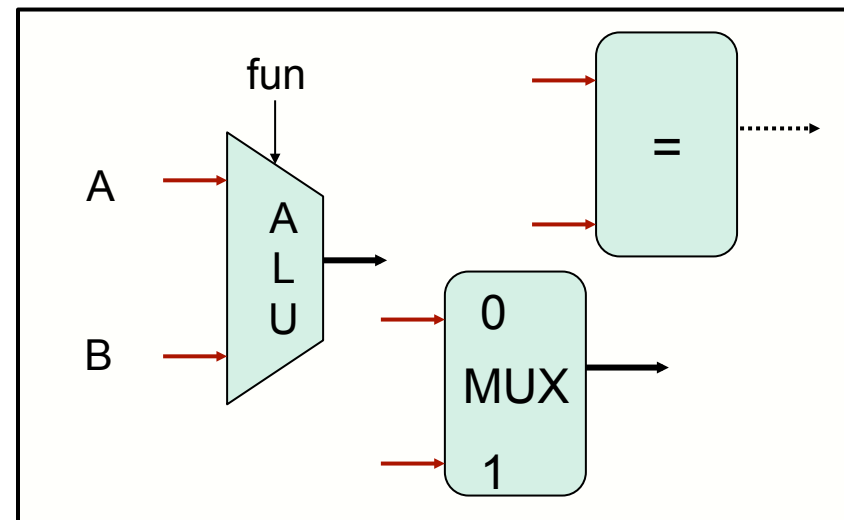
Byte	0	1	2	3	4	5	6	7	
halt	0	0							
nop	1	0							
cmovXX rA, rB	2	fn	rA	rB					
irmovq V, rB	3	0	F	rB	V				
rmmovq rA, D(rB)	4	0	rA	rB	D				
mrmmovq D(rB), rA	5	0	rA	rB	D				
OPq rA, rB	6	fn	rA	rB					
jXX Dest	7	fn	Dest						
call Dest	8	0	Dest						
ret	9	0							
pushq rA	A	0	rA	F					
popq rA	B	0	rA	F					

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

Recall: Building Blocks

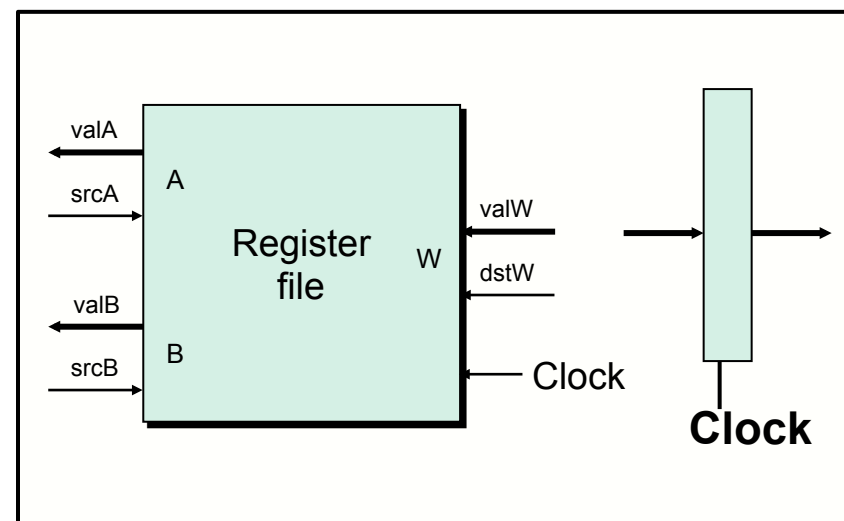
■ Combinational Logic

- Compute Boolean functions of inputs
- Continuously respond to input changes
- Operate on data and implement control



■ Storage Elements

- Store bits
- Addressable memories
- Non-addressable registers
- Loaded only as clock rises



Recall: 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 ;`

Recall: 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

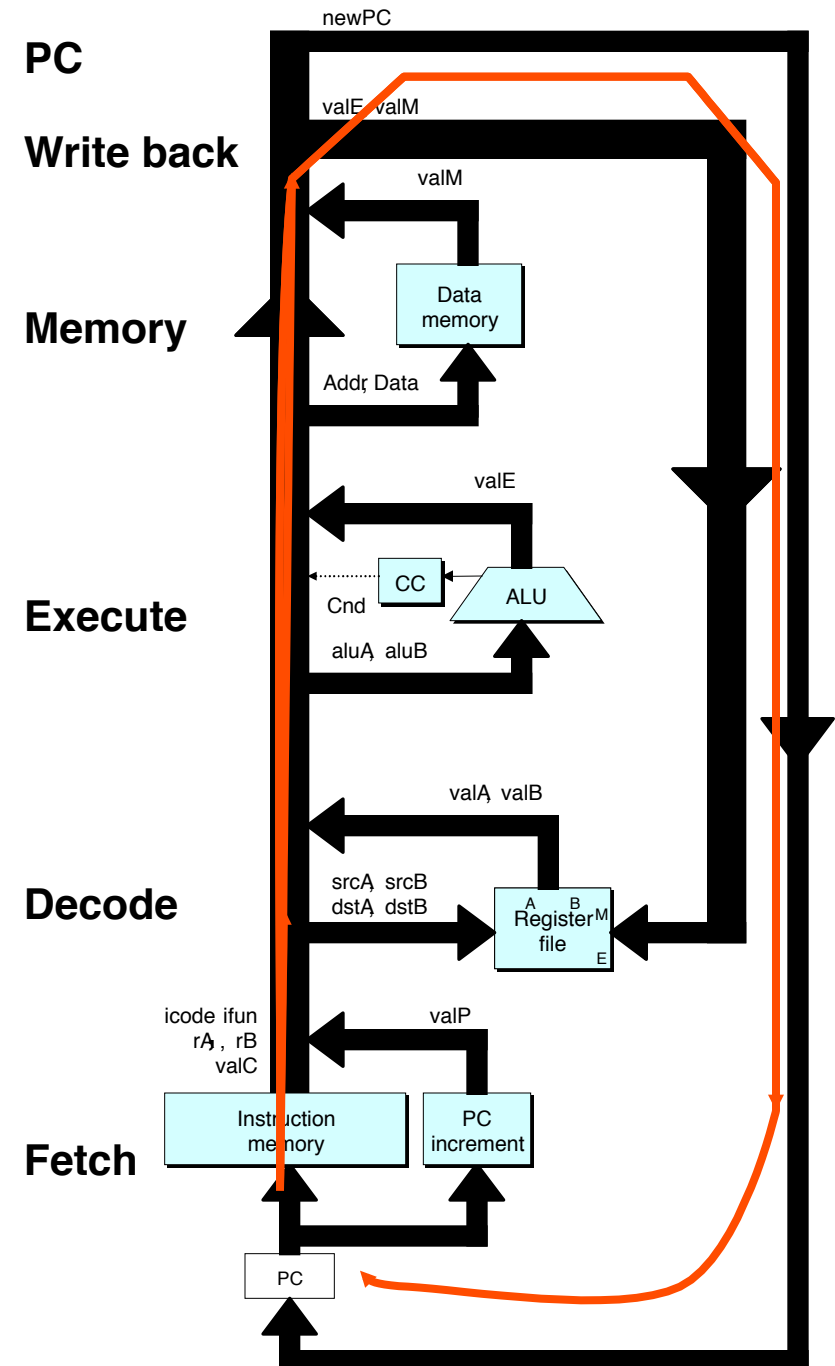
SEQ Hardware Structure

■ 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

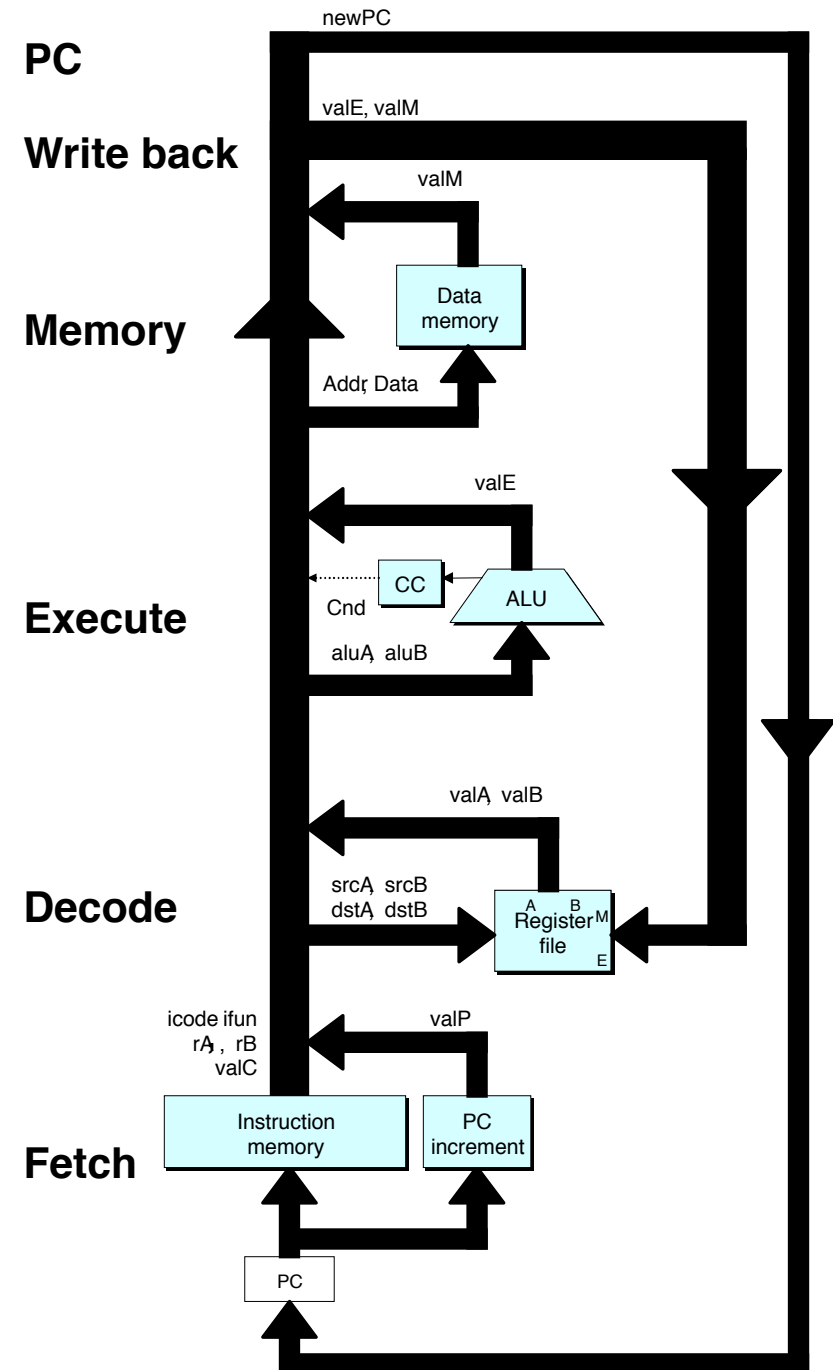
■ Instruction Flow

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

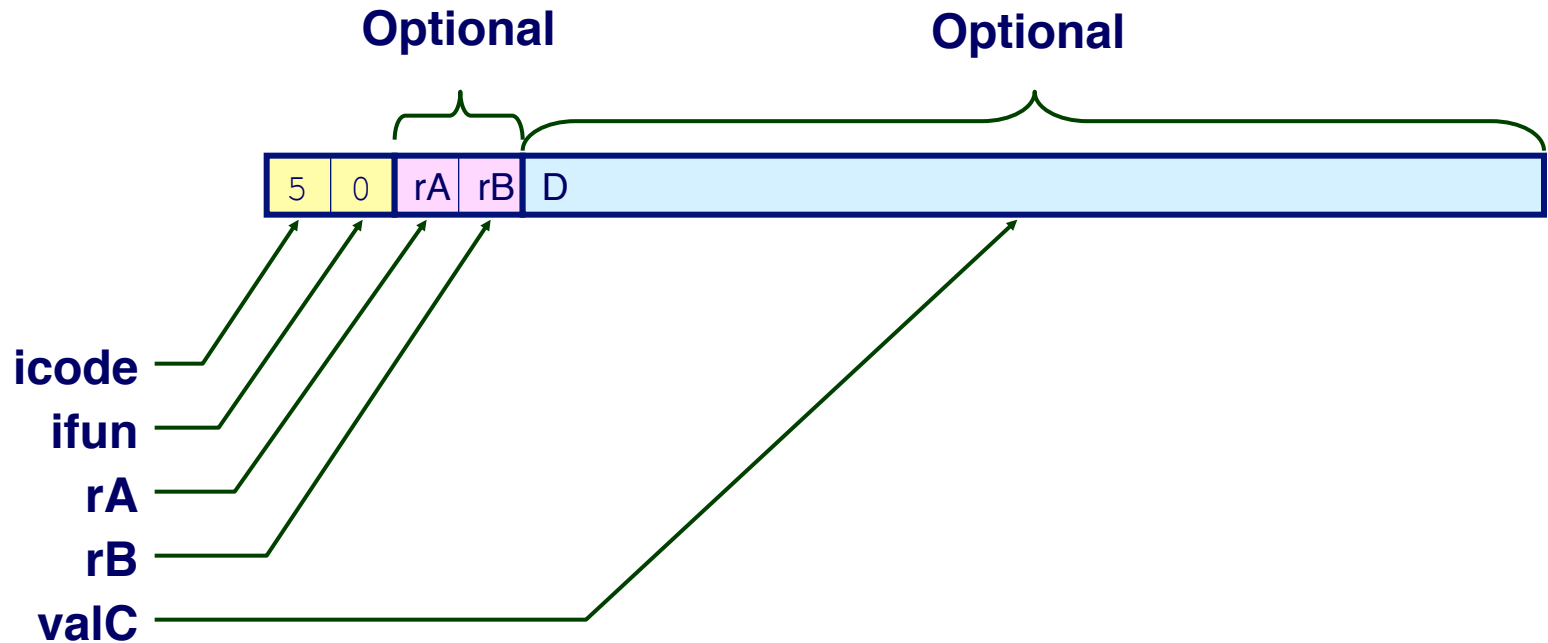


SEQ Stages

- **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



Instruction Decoding



■ Instruction Format

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

Executing Arith./Logical Operation

OP_q 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

Stage Computation: Arith/Log. Ops

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

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

Executing `rmmovq`

`rmmovq rA, D(rB)`

4	0	rA	rB
---	---	----	----

D

■ Fetch

- Read 10 bytes

■ Decode

- Read operand registers

■ Execute

- Compute effective address

■ Memory

- Write to memory

■ Write back

- Do nothing

■ PC Update

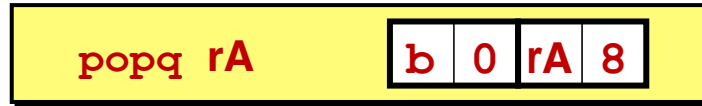
- Increment PC by 10

Stage Computation: `rmmovq`

	<code>rmmovq rA, D(rB)</code>	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valC} \leftarrow M_8[\text{PC}+2]$ $\text{valP} \leftarrow \text{PC}+10$	Read instruction byte Read register byte Read displacement D Compute next PC
Decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	Read operand A Read operand B
Execute	$\text{valE} \leftarrow \text{valB} + \text{valC}$	Compute effective address
Memory	$M_8[\text{valE}] \leftarrow \text{valA}$	Write value to memory
Write back		
PC update	$\text{PC} \leftarrow \text{valP}$	Update PC

- Use ALU for address computation

Executing popq



■ Fetch

- Read 2 bytes

■ Decode

- Read stack pointer

■ Execute

- Increment stack pointer by 8

■ Memory

- Read from old stack pointer

■ Write back

- Update stack pointer
- Write result to register

■ PC Update

- Increment PC by 2

Stage Computation: popq

	popq rA	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+2$	Read instruction byte Read register byte Compute next PC
Decode	$\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$	Read stack pointer Read stack pointer
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Increment stack pointer
Memory	$\text{valM} \leftarrow M_8[\text{valA}]$	Read from stack
Write back	$R[\%rsp] \leftarrow \text{valE}$ $R[\text{rA}] \leftarrow \text{valM}$	Update stack pointer Write back result
PC update	$\text{PC} \leftarrow \text{valP}$	Update PC

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

Executing Conditional Moves

`cmovXX rA, rB`

2	fn	rA	rB
---	----	----	----

■ Fetch

- Read 2 bytes

■ Decode

- Read operand registers

■ Execute

- If !cnd, then set destination register to 0xF

■ Memory

- Do nothing

■ Write back

- Update register (or not)

■ PC Update

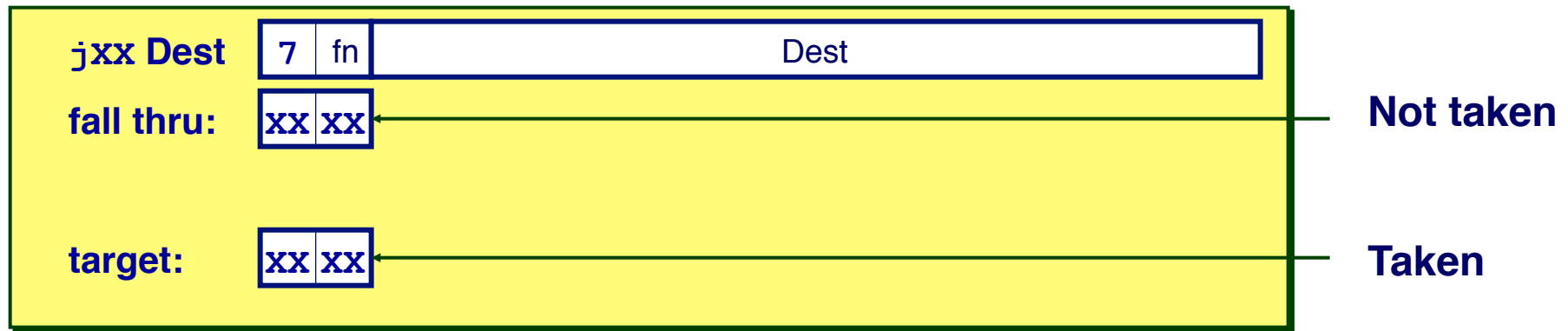
- Increment PC by 2

Stage Computation: Cond. Move

	cmovXX rA, rB	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+2$	Read instruction byte Read register byte Compute next PC
Decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow 0$	Read operand A
Execute	$\text{valE} \leftarrow \text{valB} + \text{valA}$ If ! Cond(CC,ifun) $\text{rB} \leftarrow 0xF$	Pass valA through ALU (Disable register update)
Memory		
Write back	$R[\text{rB}] \leftarrow \text{valE}$	Write back result
PC update	$\text{PC} \leftarrow \text{valP}$	Update PC

- Read register rA and pass through ALU
- Cancel move by setting destination register to 0xF
 - If condition codes & move condition indicate no move

Executing Jumps



■ Fetch

- Read 9 bytes
- Increment PC by 9

■ 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

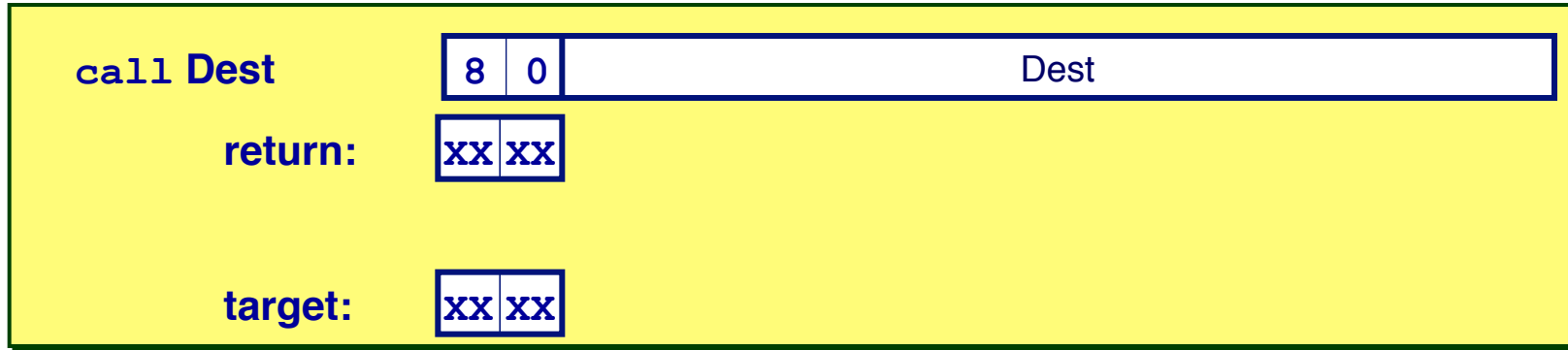
- Set PC to Dest if branch taken or to incremented PC if not branch

Stage Computation: Jumps

	jXX Dest	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Read instruction byte
	$\text{valC} \leftarrow M_8[\text{PC}+1]$	Read destination address
	$\text{valP} \leftarrow \text{PC}+9$	Fall through address
Decode		
Execute	$\text{Cnd} \leftarrow \text{Cond}(\text{CC}, \text{ifun})$	Take branch?
Memory		
Write back		
PC update	$\text{PC} \leftarrow \text{Cnd} ? \text{valC} : \text{valP}$	Update PC

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

Executing call



■ Fetch

- Read 9 bytes
- Increment PC by 9

■ Decode

- Read stack pointer

■ Execute

- Decrement stack pointer by 8

■ Memory

- Write incremented PC to new value of stack pointer

■ Write back

- Update stack pointer

■ PC Update

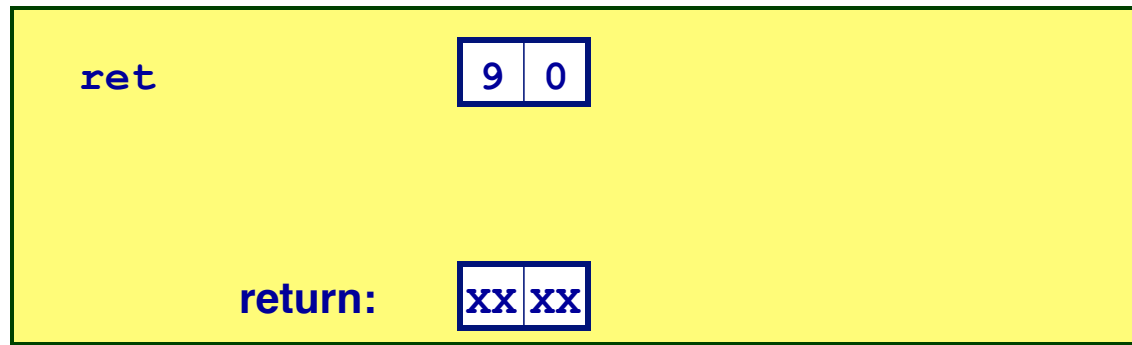
- Set PC to Dest

Stage Computation: call

	call Dest	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_8[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+9$	Read instruction byte Read destination address Compute return point
Decode	$\text{valB} \leftarrow R[\%rsp]$	Read stack pointer
Execute	$\text{valE} \leftarrow \text{valB} + -8$	Decrement stack pointer
Memory	$M_8[\text{valE}] \leftarrow \text{valP}$	Write return value on stack
Write back	$R[\%rsp] \leftarrow \text{valE}$	Update stack pointer
PC update	$\text{PC} \leftarrow \text{valC}$	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 8

■ Memory

- Read return address from old stack pointer

■ Write back

- Update stack pointer

■ PC Update

- Set PC to return address

Stage Computation: `ret`

	<code>ret</code>	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Read instruction byte
Decode	$\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$	Read operand stack pointer Read operand stack pointer
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Increment stack pointer
Memory	$\text{valM} \leftarrow M_8[\text{valA}]$	Read return address
Write back	$R[\%rsp] \leftarrow \text{valE}$	Update stack pointer
PC update	$\text{PC} \leftarrow \text{valM}$	Set PC to return address

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

Computation Steps

		OPq rA, rB
Fetch	icode,ifun	icode:ifun $\leftarrow M_1[PC]$
	rA,rB	rA:rB $\leftarrow M_1[PC+1]$
	valC	
	valP	valP $\leftarrow PC+2$
Decode	valA, srcA	valA $\leftarrow R[rA]$
	valB, srcB	valB $\leftarrow R[rB]$
Execute	valE	valE $\leftarrow \text{valB OP valA}$
	Cond code	Set CC
Memory	valM	
Write back	dstE	R[rB] $\leftarrow \text{valE}$
	dstM	
PC update	PC	PC $\leftarrow \text{valP}$

Read instruction byte
 Read register byte
 [Read constant word]
 Compute next PC
 Read operand A
 Read operand B
 Perform ALU operation
 Set/use cond. code reg
 [Memory read/write]
 Write back ALU result
 [Write back memory result]
 Update PC

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

Computation Steps

		call Dest
Fetch	icode,ifun	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_8[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+9$
	rA,rB	
	valC	
	valP	
Decode	valA, srcA	$\text{valB} \leftarrow R[\%rsp]$
	valB, srcB	
Execute	valE	$\text{valE} \leftarrow \text{valB} + -8$
	Cond code	
Memory	valM	$M_8[\text{valE}] \leftarrow \text{valP}$
Write back	dstE	$R[\%rsp] \leftarrow \text{valE}$
	dstM	
PC update	PC	$\text{PC} \leftarrow \text{valC}$

Read instruction byte
 [Read register byte]
 Read constant word
 Compute next PC
 [Read operand A]
 Read operand B
 Perform ALU operation
 [Set /use cond. code reg]
 Memory read/write
 Write back ALU result
 [Write back memory result]
 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
Cnd	Branch/move flag

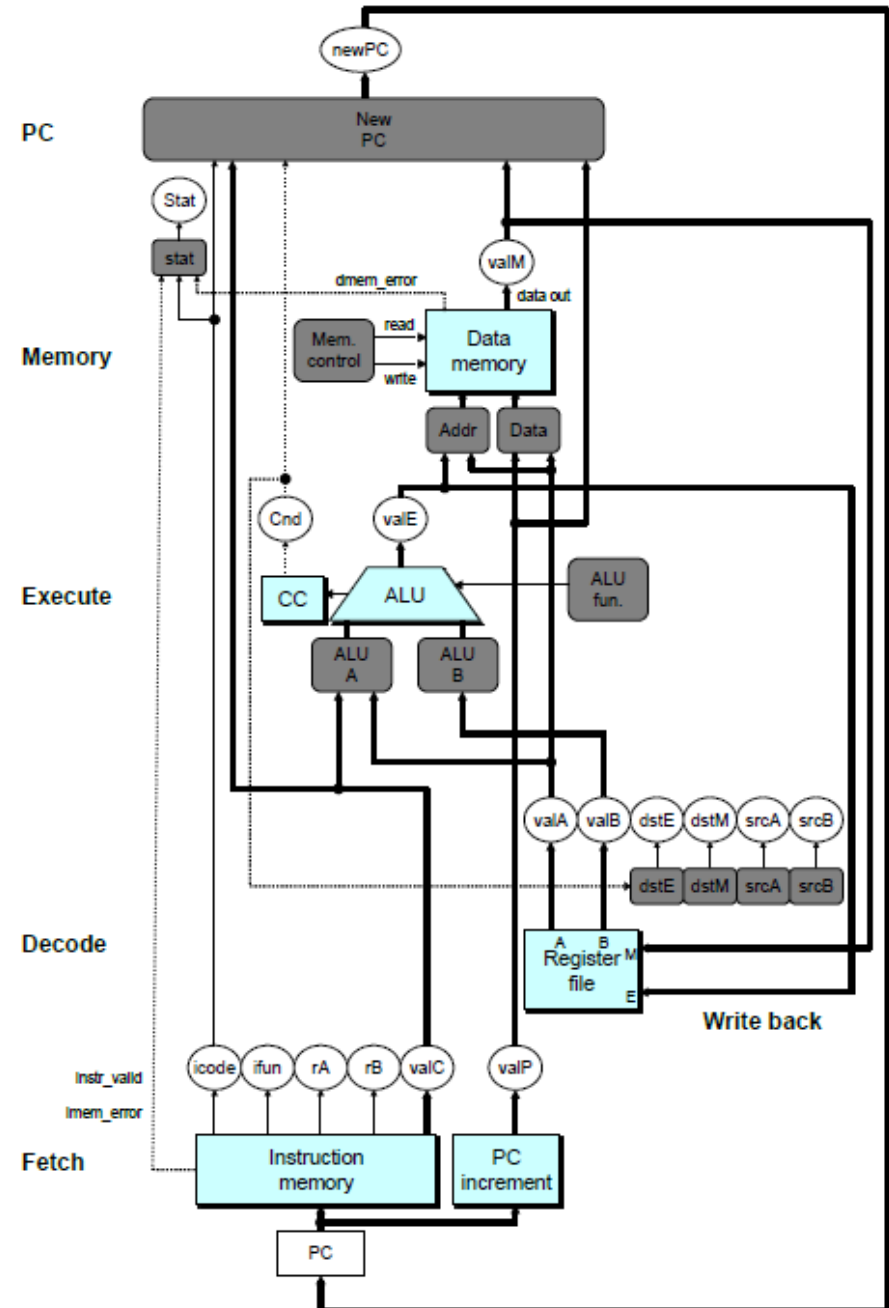
■Memory

valM	Value from memory
------	-------------------

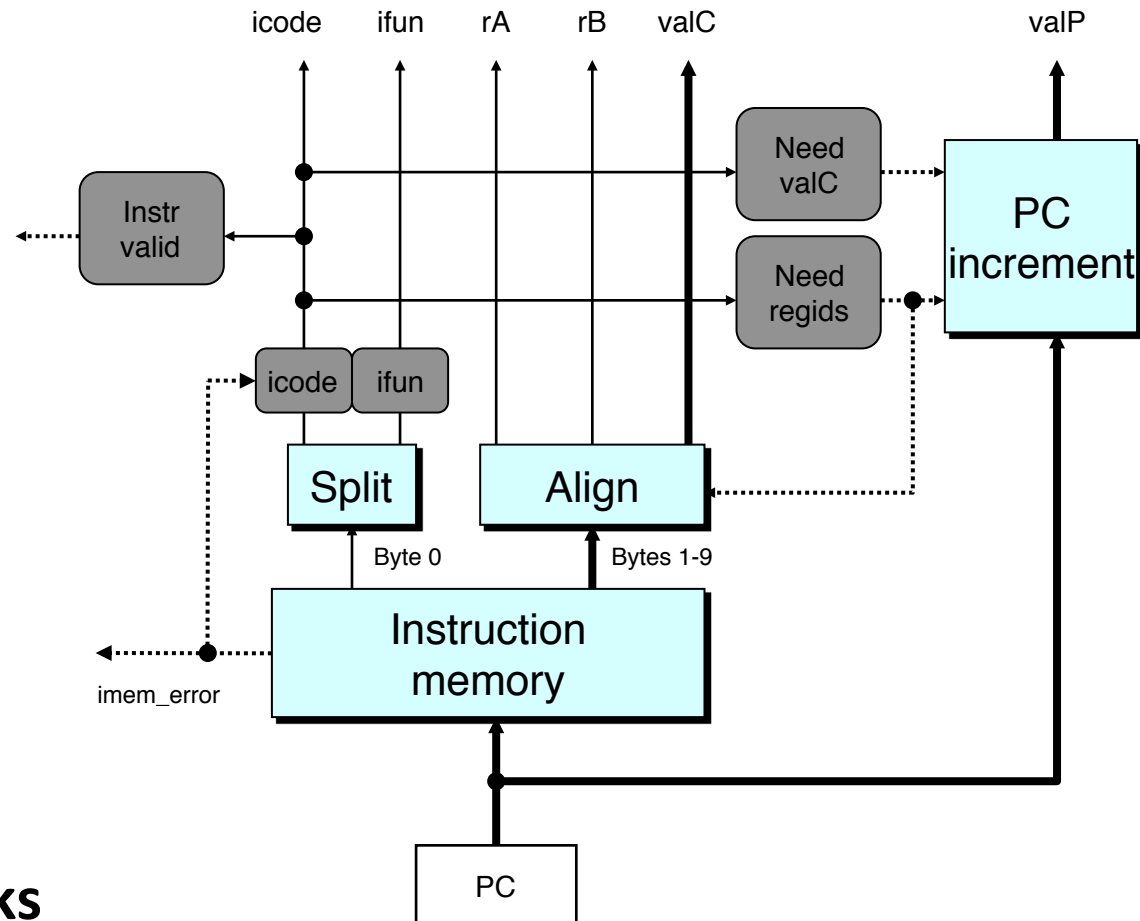
SEQ Hardware

■ Key

- Blue boxes: predefined hardware blocks
 - E.g., memories, ALU
- Gray boxes: control logic
 - Describe in HCL
- White ovals: labels for signals
- Thick lines: 64-bit word values
- Thin lines: 4-8 bit values
- Dotted lines: 1-bit values



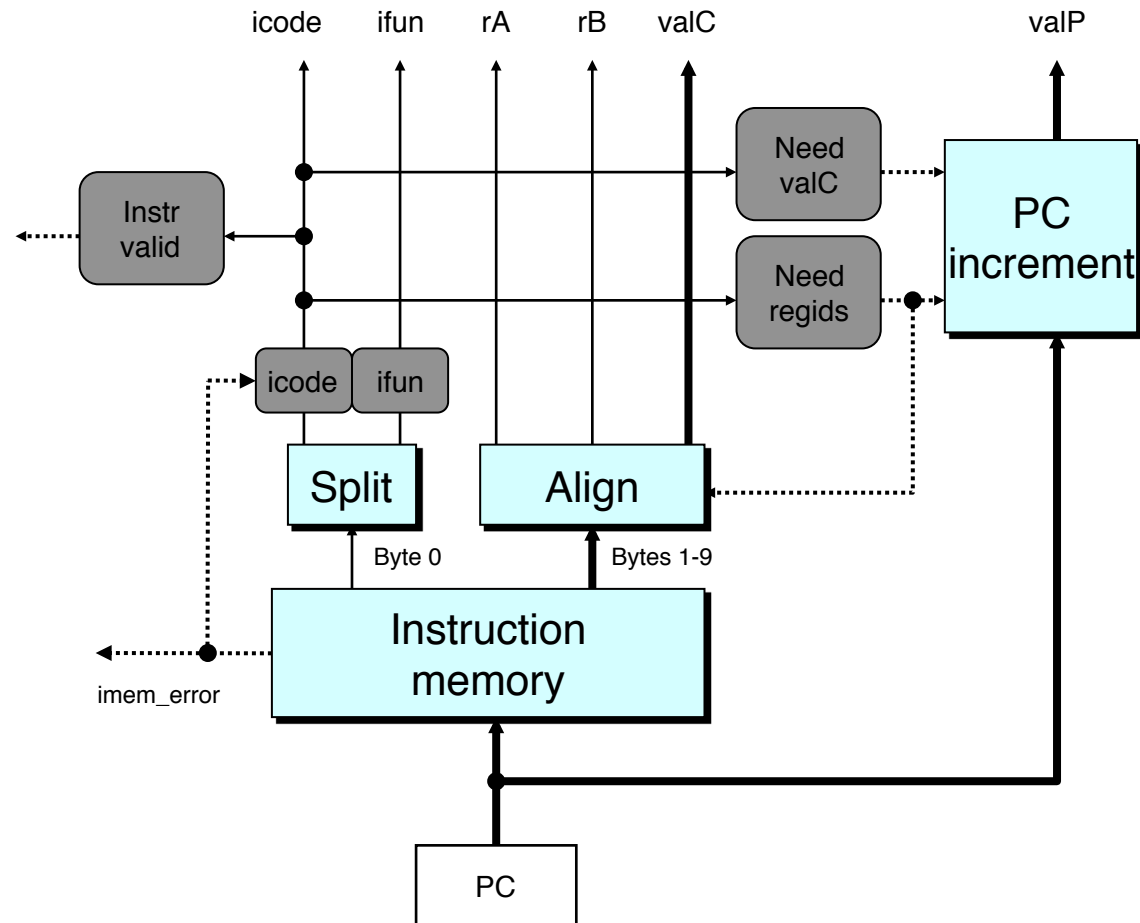
Fetch Logic



■ Predefined Blocks

- **PC:** Register containing PC
- **Instruction memory:** Read 10 bytes (PC to PC+9)
 - Signal invalid address
- **Split:** Divide instruction byte into icode and ifun
- **Align:** Get fields for rA, rB, and valC

Fetch Logic



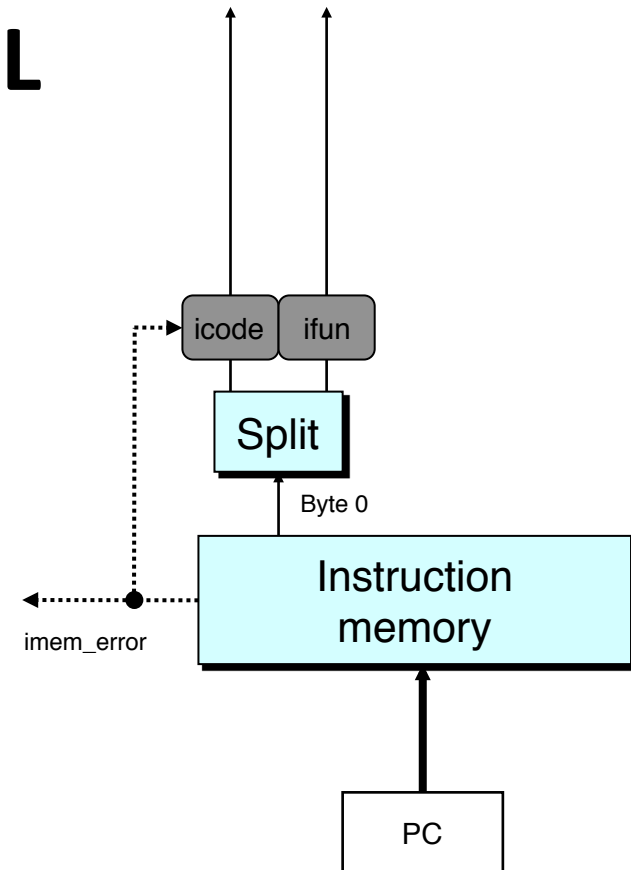
■ Control Logic

- Instr. Valid: Is this instruction valid?
- icode, ifun: Generate no-op if invalid address
- Need regids: Does this instruction have a register byte?
- Need valC: Does this instruction have a constant word?

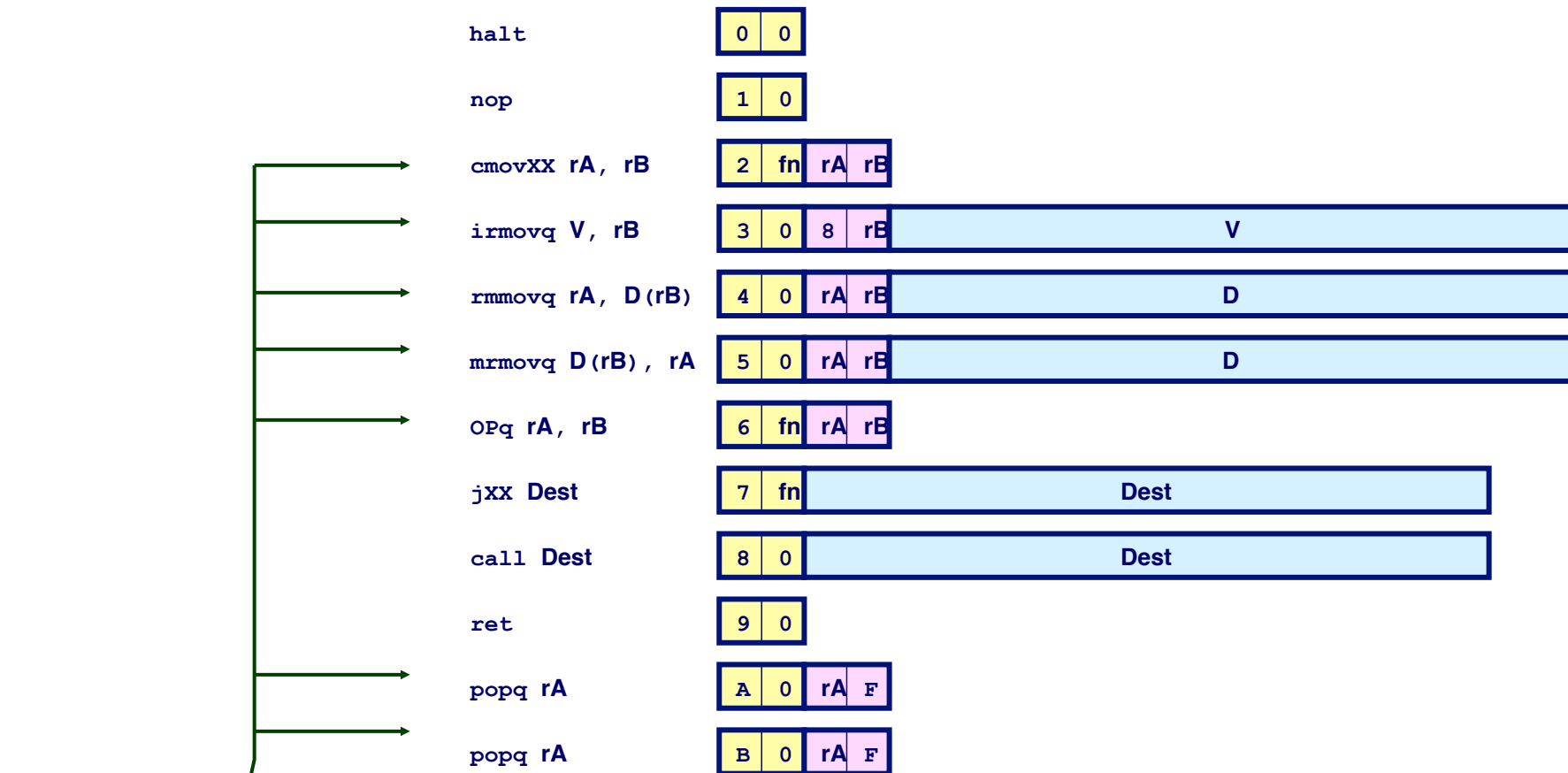
Fetch Control Logic in HCL

```
# Determine instruction code
int icode = [
    imem_error: INOP;
    1: imem_icode;
];

# Determine instruction function
int ifun = [
    imem_error: FNONE;
    1: imem_ifun;
];
```



Fetch Control Logic in HCL



```
bool need_regids =
    icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
               IIRMOVQ, IRMMOVQ, IMRMVQ };
```

```
bool instr_valid = icode in
    { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMVQ,
      IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ };
```

Decode Logic

■ Register File

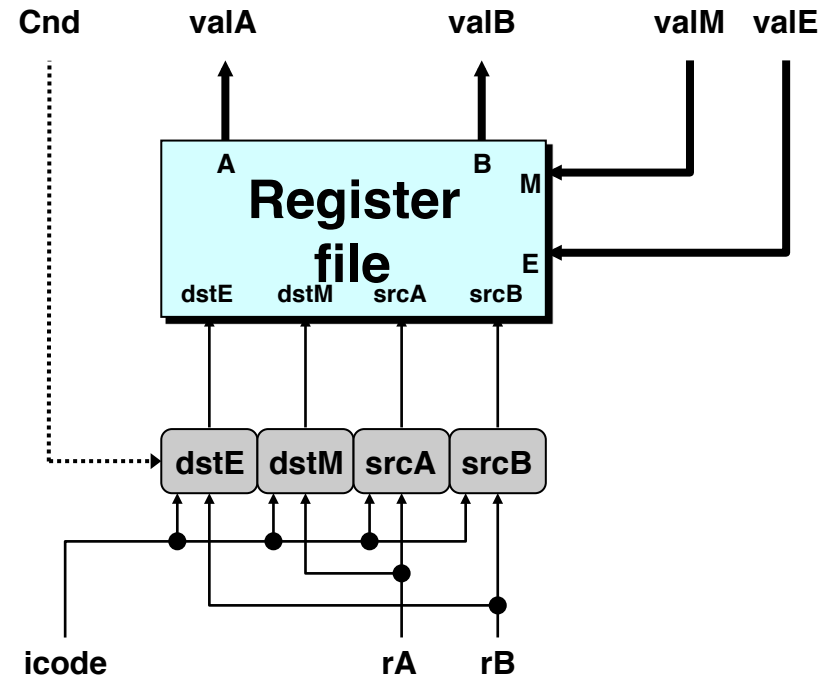
- Read ports A, B
- Write ports E, M
- Addresses are register IDs or 15 (0xF) (no access)

■ Control Logic

- srcA, srcB: read port addresses
- dstE, dstM: write port addresses

■ Signals

- Cnd: Indicate whether or not to perform conditional move
 - Computed in Execute stage



A Source

	OPq rA, rB	
Decode	valA \leftarrow R[rA]	Read operand A
	cmovXX rA, rB	
Decode	valA \leftarrow R[rA]	Read operand A
	rmmovq rA, D(rB)	
Decode	valA \leftarrow R[rA]	Read operand A
	popq rA	
Decode	valA \leftarrow R[%rsp]	Read stack pointer
	jXX Dest	
Decode		No operand
	call Dest	
Decode		No operand
	ret	
Decode	valA \leftarrow R[%rsp]	Read stack pointer

```

int srcA = [
    icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ } : rA;
    icode in { IPOPOPQ, IRET } : RRSP;
    1 : RNONE; # Don't need register
];

```

E Destination

	OPq rA, rB	
Write-back	R[rB] ← valE	Write back result
	cmovXX rA, rB	
Write-back	R[rB] ← valE	Conditionally write back result
	rmmovq rA, D(rB)	
Write-back		None
	popq rA	
Write-back	R[%rsp] ← valE	Update stack pointer
	jXX Dest	
Write-back		None
	call Dest	
Write-back	R[%rsp] ← valE	Update stack pointer
	ret	
Write-back	R[%rsp] ← valE	Update stack pointer

```

int dstE = [
    icode in { IRRMOVQ } && Cnd : rB;
    icode in { IIRMOVQ, IOPQ } : rB;
    icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE; # Don't write any register
];

```

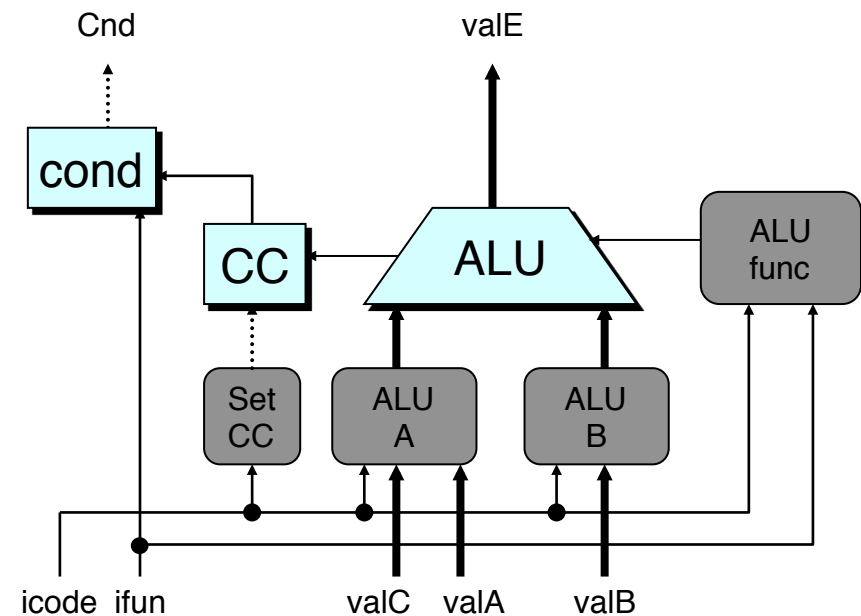
Execute Logic

■ Units

- ALU
 - Implements 4 required functions
 - Generates condition code values
- CC
 - Register with 3 condition code bits
- cond
 - Computes conditional jump/move 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

	OPq rA, rB	
Execute	$\text{valE} \leftarrow \text{valB} \text{ OP } \text{valA}$	Perform ALU operation
	cmovXX rA, rB	
Execute	$\text{valE} \leftarrow 0 + \text{valA}$	Pass valA through ALU
	rmmovq rA, D(rB)	
Execute	$\text{valE} \leftarrow \text{valB} + \text{valC}$	Compute effective address
	popq rA	
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Increment stack pointer
	jXX Dest	
Execute		No operation
	call Dest	
Execute	$\text{valE} \leftarrow \text{valB} + -8$	Decrement stack pointer
	ret	
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Increment stack pointer

```

int aluA = [
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMVQ } : valC;
    icode in { ICALL, IPUSHQ } : -8;
    icode in { IRET, IPOPOPQ } : 8;
    # Other instructions don't need ALU

```

ALU Operation

	OPI rA, rB	
Execute	$\text{valE} \leftarrow \text{valB} \text{ OP } \text{valA}$	Perform ALU operation
	cmovXX rA, rB	
Execute	$\text{valE} \leftarrow 0 + \text{valA}$	Pass valA through ALU
	rmmovl rA, D(rB)	
Execute	$\text{valE} \leftarrow \text{valB} + \text{valC}$	Compute effective address
	popq rA	
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Increment stack pointer
	jXX Dest	
Execute		No operation
	call Dest	
Execute	$\text{valE} \leftarrow \text{valB} + -8$	Decrement stack pointer
	ret	
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Increment stack pointer

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

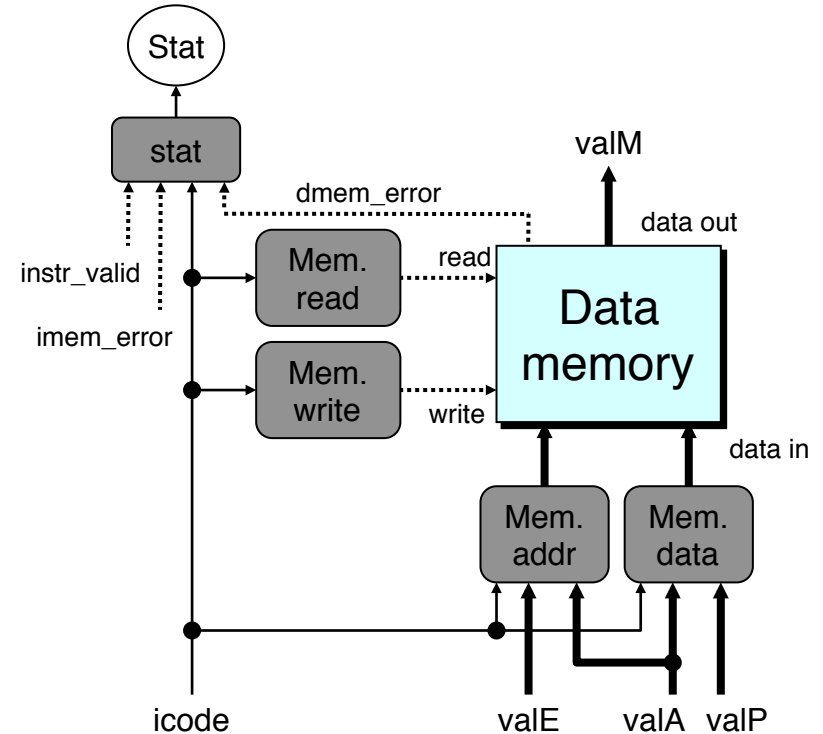

Memory Logic

■ Memory

- Reads or writes memory word

■ Control Logic

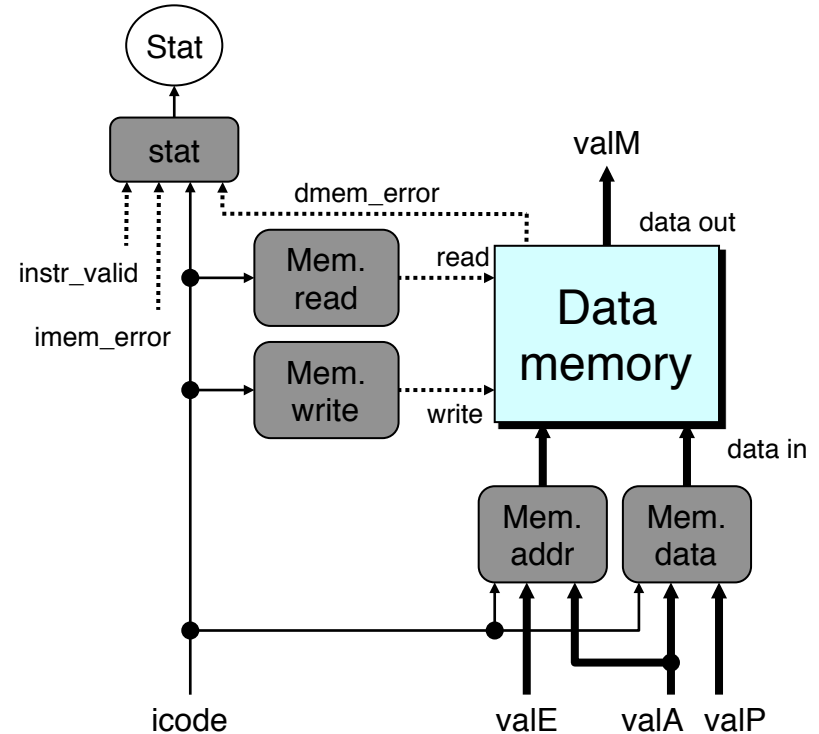
- stat: What is instruction status?
- Mem. read: should word be read?
- Mem. write: should word be written?
- Mem. addr.: Select address
- Mem. data.: Select data



Instruction Status

■ Control Logic

- stat: What is instruction status?



```
## Determine instruction status
int Stat = [
    imem_error || dmem_error : SADR;
    !instr_valid: SINS;
    icode == IHALT : SHLT;
    1 : SAOK;
];
```

Memory Address

	OPq rA, rB	
Memory		No operation
	rmmovq rA, D(rB)	
Memory	$M_8[\text{valE}] \leftarrow \text{valA}$	Write value to memory
	popq rA	
Memory	$\text{valM} \leftarrow M_8[\text{valA}]$	Read from stack
	jXX Dest	
Memory		No operation
	call Dest	
Memory	$M_8[\text{valE}] \leftarrow \text{valP}$	Write return value on stack
	ret	
Memory	$\text{valM} \leftarrow M_8[\text{valA}]$	Read return address

```

int mem_addr = [
    icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
    icode in { IPOPOPQ, IRET } : valA;
    # Other instructions don't need address

```

```

];

```

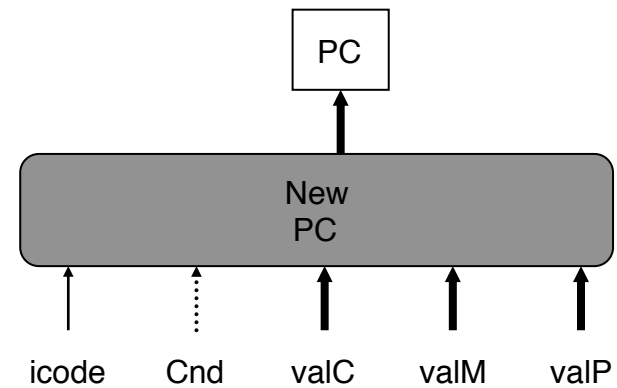
Memory Read

	OPq rA, rB	
Memory		No operation
	rmmovq rA, D(rB)	
Memory	$M_8[valE] \leftarrow valA$	Write value to memory
	popq rA	
Memory	$valM \leftarrow M_8[valA]$	Read from stack
	jXX Dest	
Memory		No operation
	call Dest	
Memory	$M_8[valE] \leftarrow valP$	Write return value on stack
	ret	
Memory	$valM \leftarrow M_8[valA]$	Read return address

```
bool mem_read = icode in { IMRMOVQ, IPOPOP, IRET };
```

PC Update Logic

- **New PC**
 - Select next value of PC



PC Update

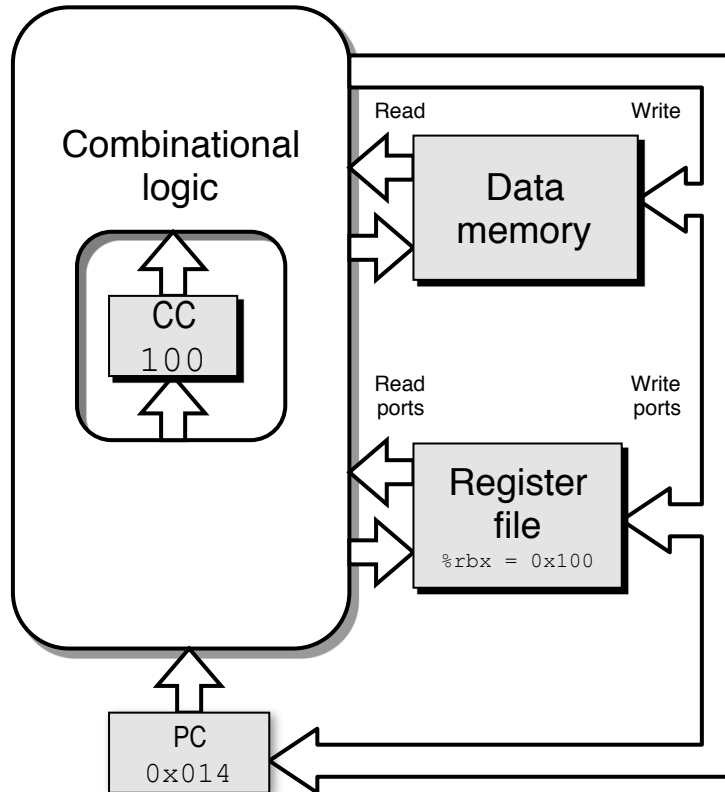
	OPq rA, rB	
PC update	PC \leftarrow valP	Update PC
	rmmovq rA, D(rB)	
PC update	PC \leftarrow valP	Update PC
	popq rA	
PC update	PC \leftarrow valP	Update PC
	jXX Dest	
PC update	PC \leftarrow Cnd ? valC : valP	Update PC
	call Dest	
PC update	PC \leftarrow valC	Set PC to destination
	ret	
PC update	PC \leftarrow valM	Set PC to return address

```

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

```

SEQ Operation



■ State

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

All updated as clock rises

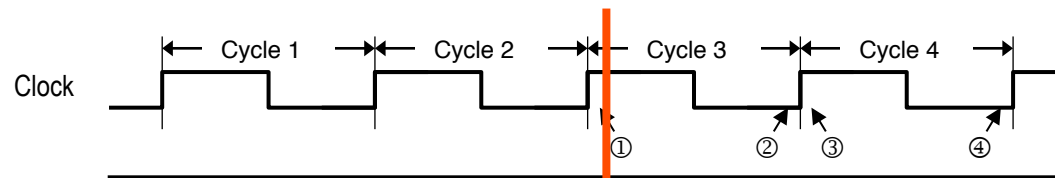
■ Combinational Logic

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

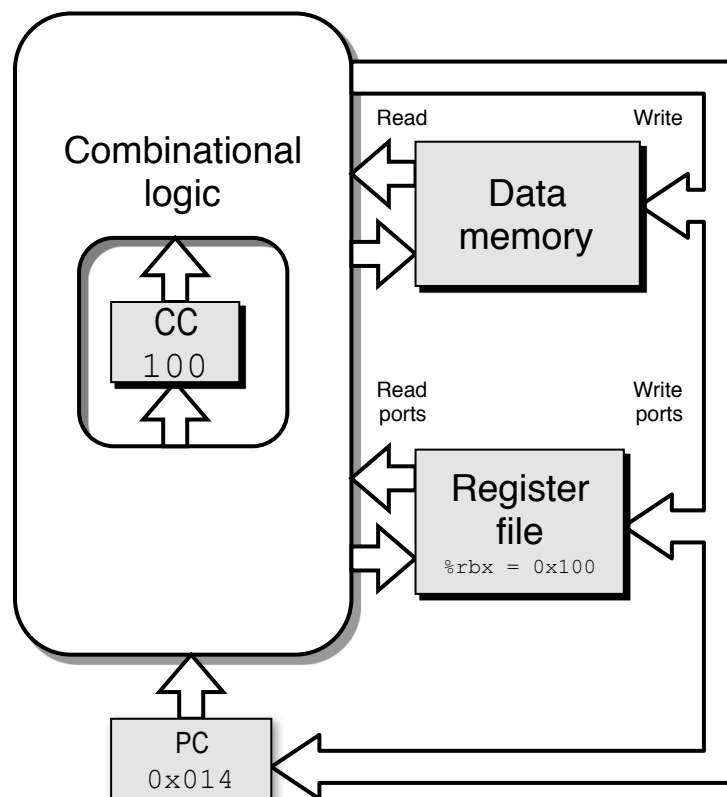
SEQ

Operation

#2



Cycle 1:	0x000:	irmovq \$0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq \$0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

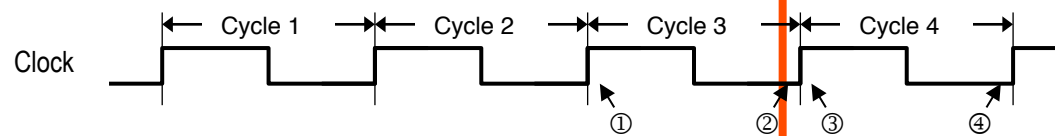


- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes

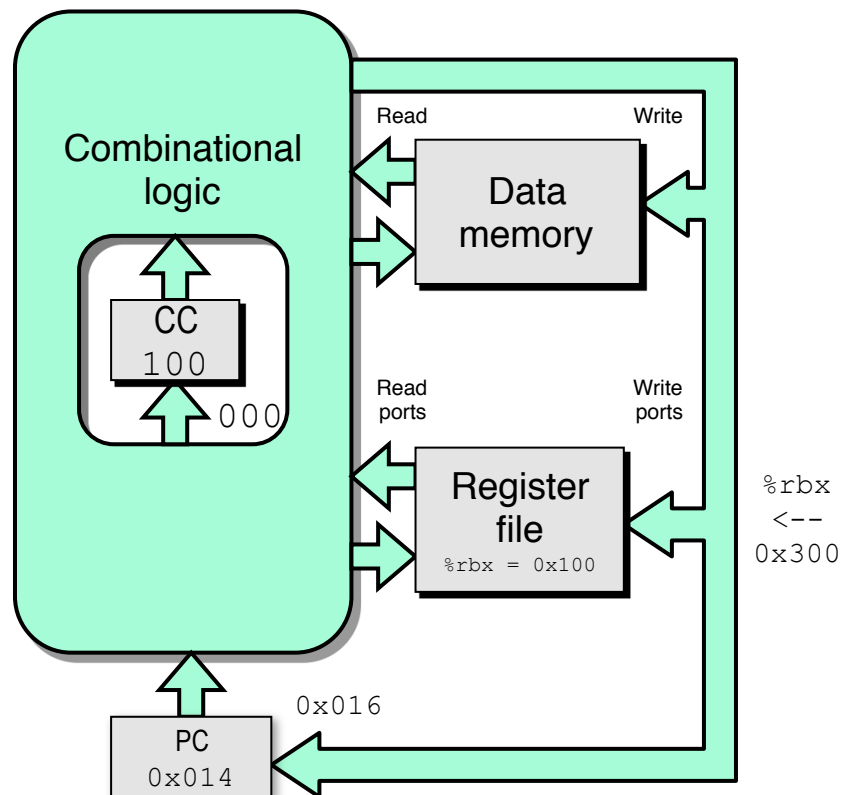
SEQ

Operation

#3



Cycle 1:	0x000: irmovq \$0x100,%rbx # %rbx <-- 0x100
Cycle 2:	0x00a: irmovq \$0x200,%rdx # %rdx <-- 0x200
Cycle 3:	0x014: addq %rdx,%rbx # %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016: je dest # Not taken
Cycle 5:	0x01f: rmmovq %rbx,0(%rdx) # M[0x200] <-- 0x300



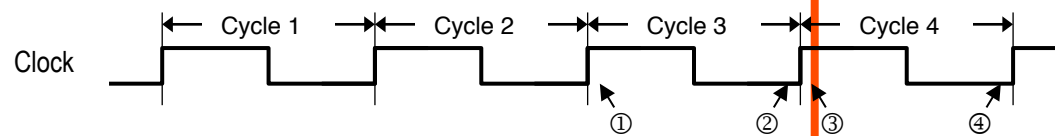
- state set according to second `irmovq` instruction
- combinational logic generates results for `addq` instruction

%rbx
<--
0x300

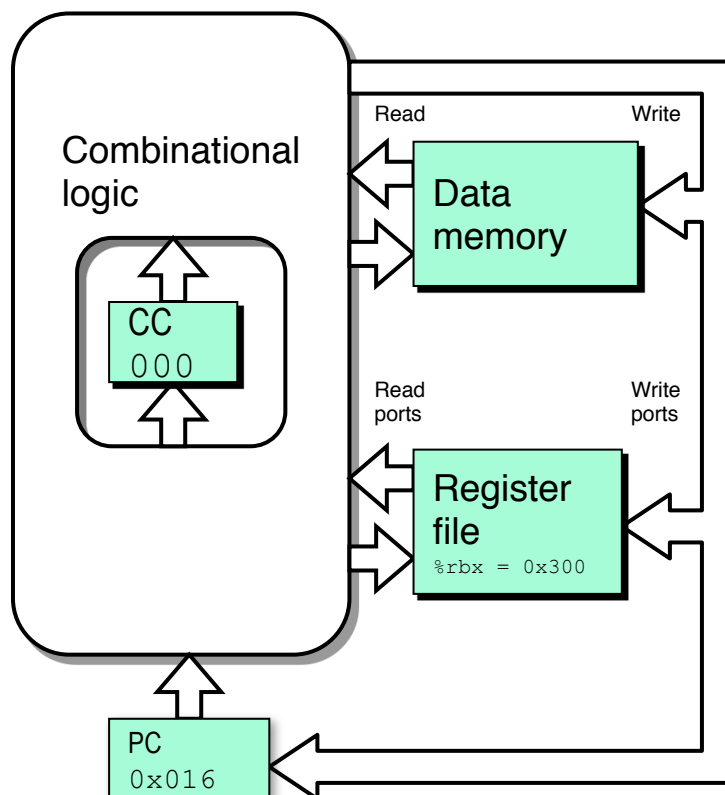
SEQ

Operation

#4



Cycle 1:	0x000:	irmovq \$0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq \$0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

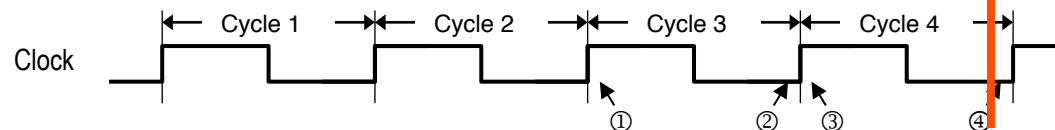


- state set according to addq instruction
- combinational logic starting to react to state changes

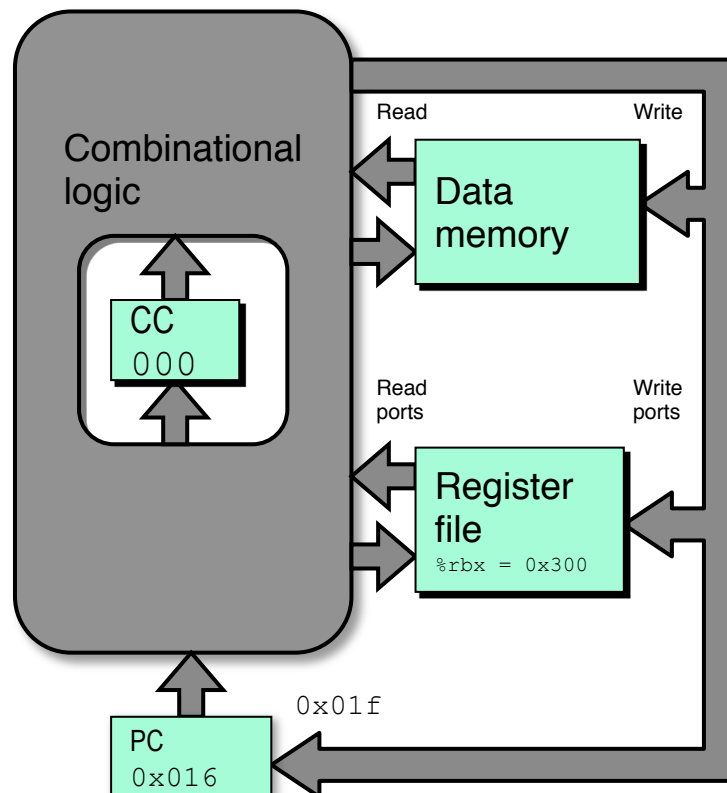
SEQ

Operation

#5



Cycle 1:	0x000:	irmovq \$0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq \$0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300



- state set according to addq 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