Processor Architecture II:
SEQ: Sequential Implementation

Introduction to Computer Systems
10th Lecture, Oct 18, 2017

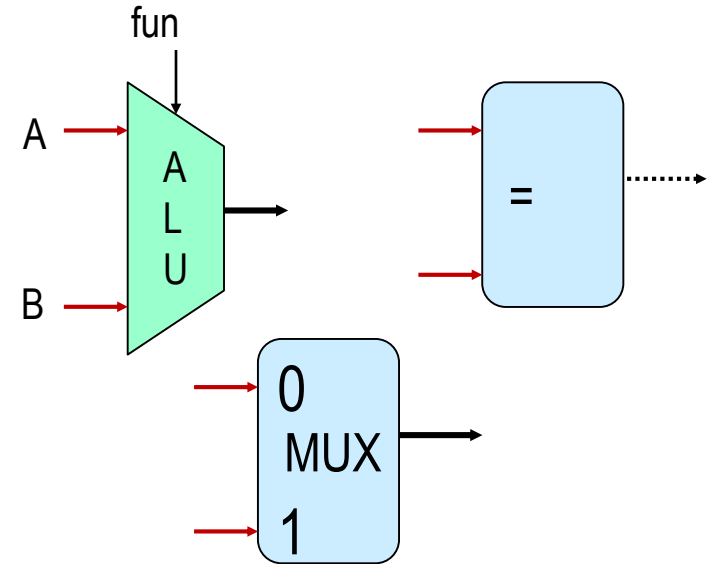**Instructors:**

Xiangqun Chen，Junlin Lu

Guangyu Sun，Xuetao Guan
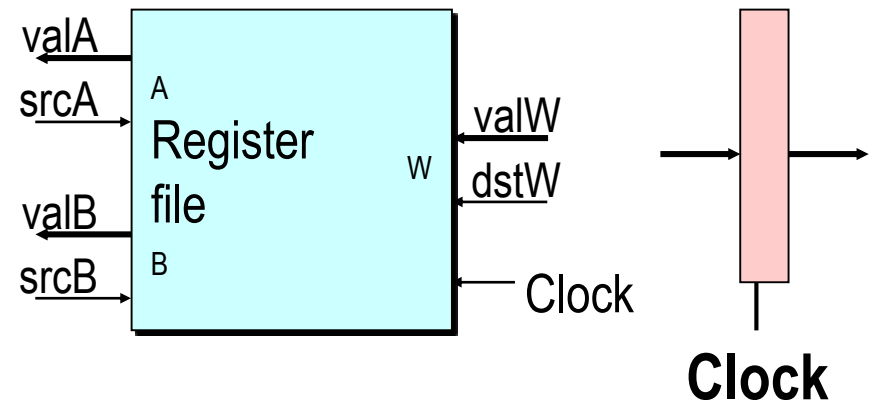
Building Blocks

- ## Combinational Logic

  - Compute Boolean functions of inputs

  - Continuously respond to input changes

  - Operate on data and implement control

- ## Storage Elements

  - Store bits

  - Addressable memories

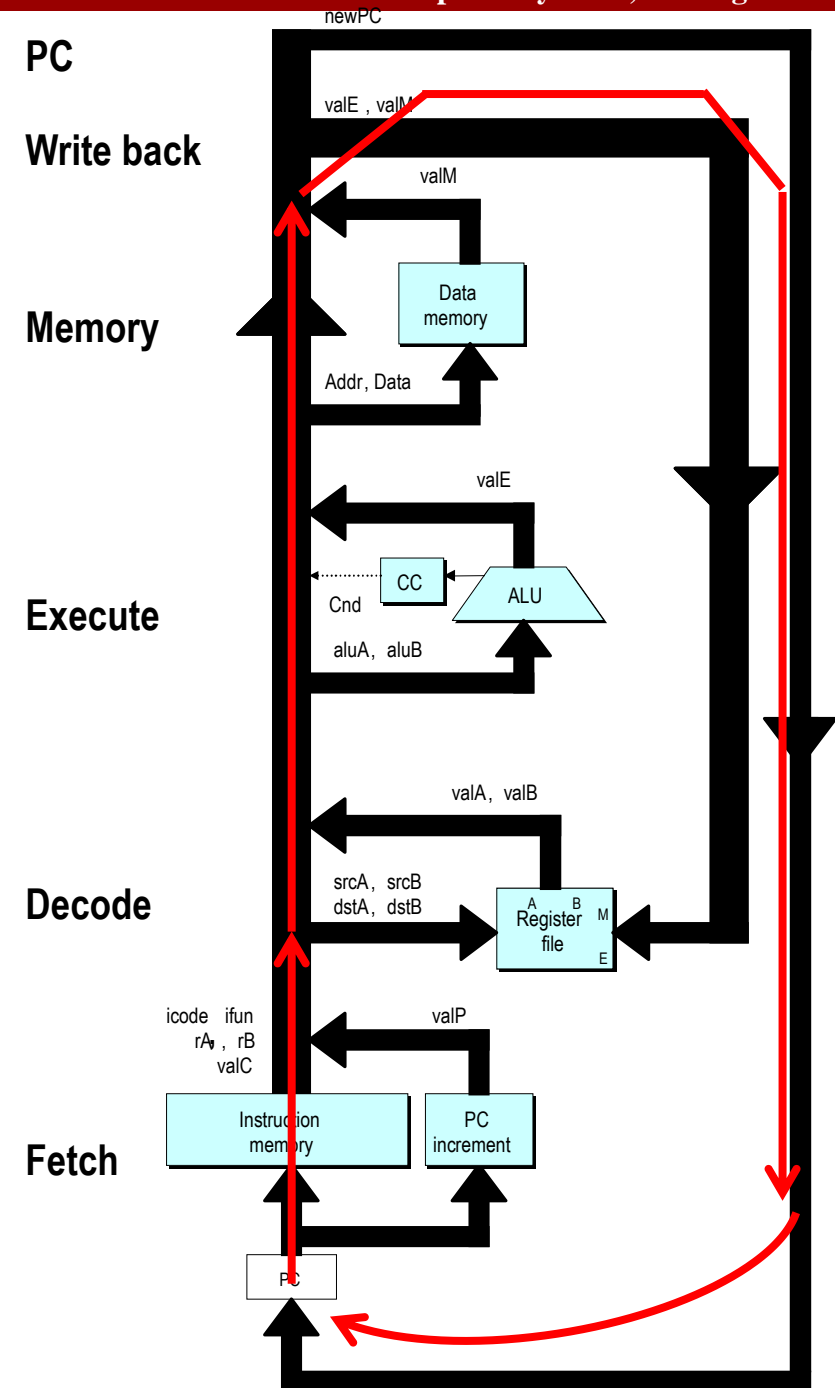 - Non-addressable registers

  - Loaded only as clock rises

fun

A

A
L
U

B

=

0
MUX
1

valA

srcA

A

Register
file

W

valB

srcB

B

valW

dstW

Clock

Clock

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

PC

newPC

valE , valM

Write back

valM

Memory

Data memory

Addr, Data

valE

CC   ALU

Cnd

Execute

aluA, aluB

valA, valB

srcA, srcB
dstA, dstB

Decode

Register file

icode ifun
rA , rB
valC

valP

Instruction memory   PC increment

Fetch

PC

SEQ Stages

- **Fetch**
  - Read instruction from instruction memory

- **Decode**
  - Read program registers
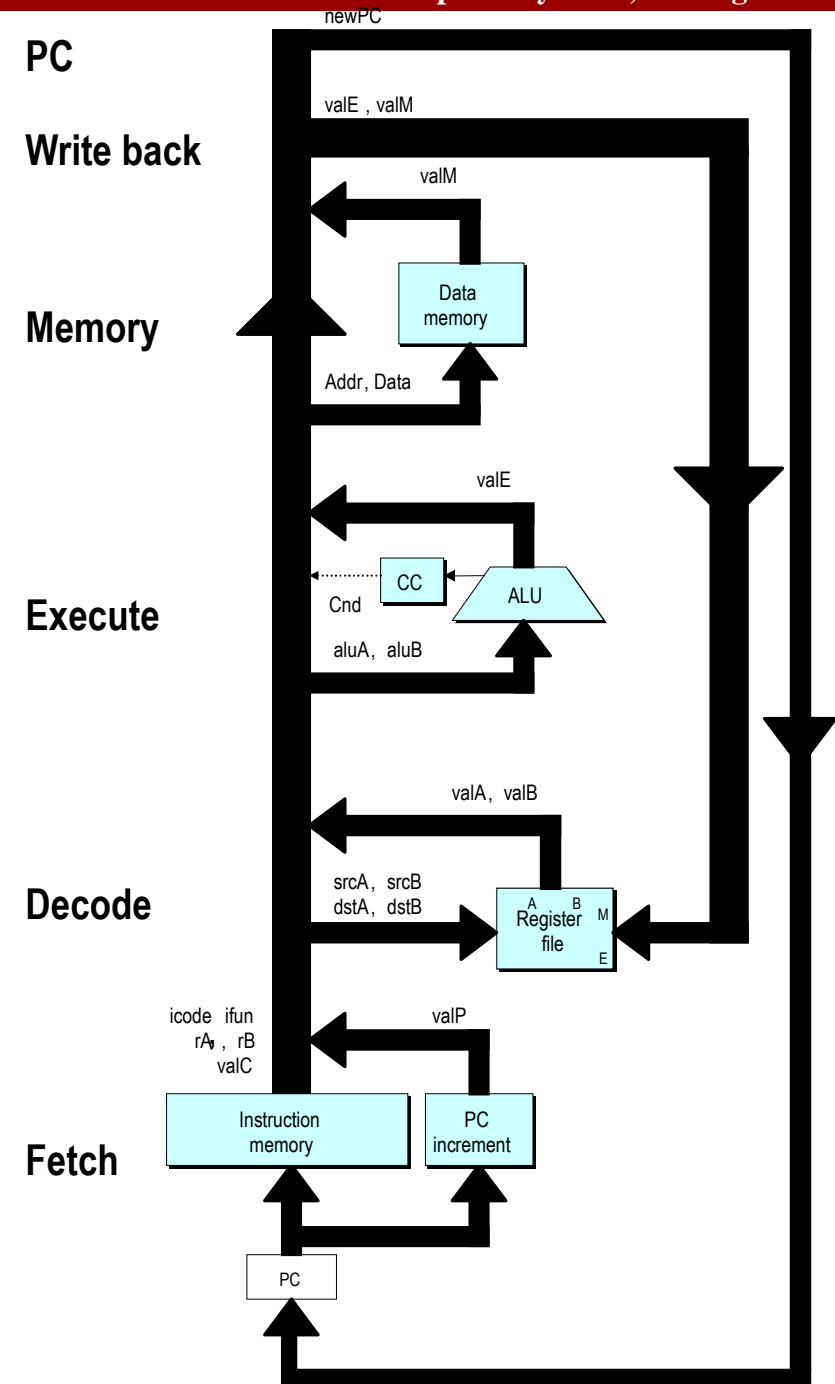
- **Execute**
  - Compute value or address

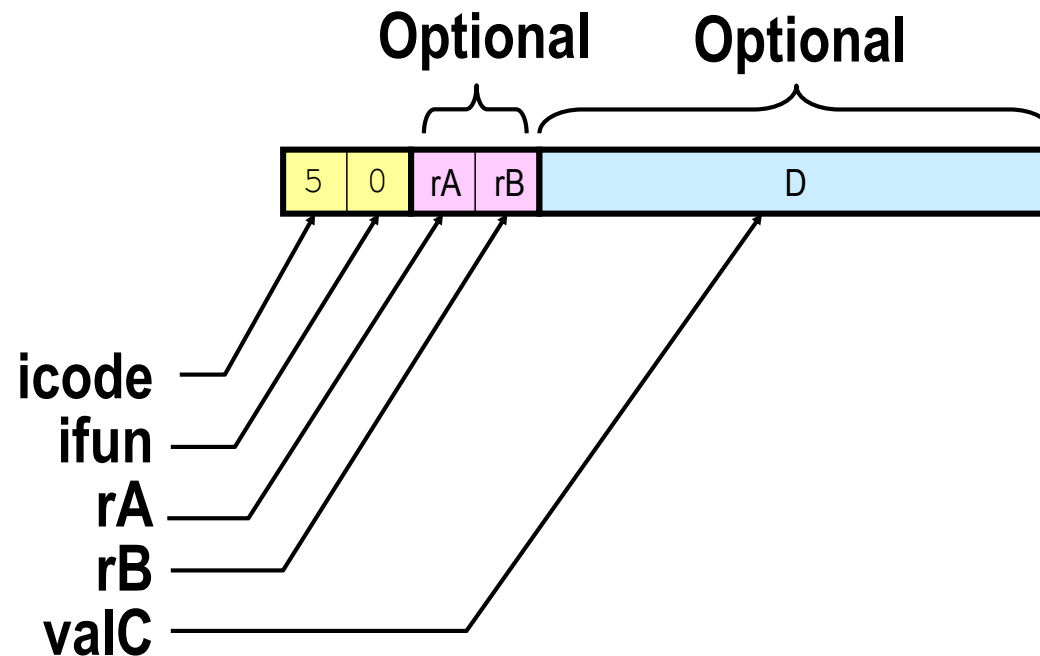- **Memory**
  - Read or write data

- **Write Back**
  - Write program registers

- **PC**
  - Update program counter
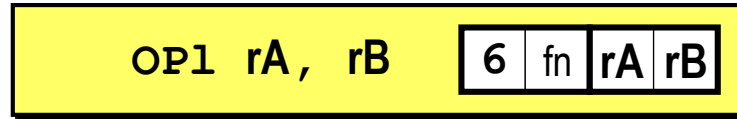

4

Instruction Decoding

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

Executing Arith./Logical Operation

| OP1 rA, rB | 6 | fn | 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

	OPl rA, rB	
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[rA]	Read operand A
	valB ← R[rB]	Read operand B
Execute	valE ← valB OP valA	Perform ALU operation
	Set CC	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**

Executing `rmmovl` `(store)`

`rmmovl rA, D(rB)`	4	0	rA	rB	D

■ Fetch

- Read 6 bytes

■ Decode

- Read operand registers

■ Execute

- Compute effective address

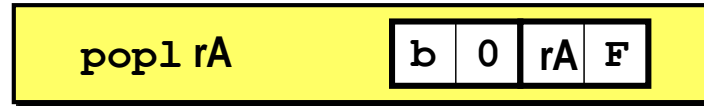### ■ Memory

- Write to memory

■ Write back

- Do nothing

■ PC Update

- Increment PC by 6

Stage Computation: `rmmovl`

	`rmmovl rA, D(rB)`	
Fetch	icode:ifun ← M₁[PC]	Read instruction byte
	rA:rB ← M₁[PC+1]	Read register byte
	valC ← M₄[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	M₄[valE] ← valA	Write value to memory
Write back		
PC update	PC ← valP	Update PC

- **Use ALU for address computation**

Executing `popl`

`popl rA`	b 0	rA F

■**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

Stage Computation: `popl`

	popl rA	
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[`%esp`] valB ← R [`%esp`]	Read stack pointer Read stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M$_1$[valA]	Read from stack
Write back	R[`%esp`] ← valE R[rA] ← valM	Update stack pointer Write back result
PC update	PC ← valP	Update PC

- **Use ALU to increment stack pointer**

- **Must update two registers**
  - Popped value
  - New stack pointer

11

Executing Jumps

jXX Dest	7	fn	Dest
fall thru:	XX	XX	← Not taken
target:	XX	XX	← Taken
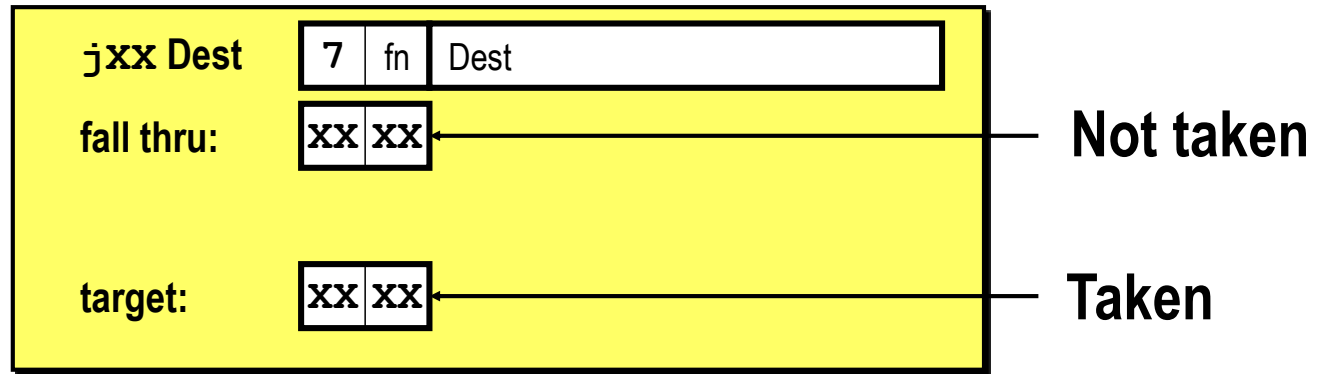
■Fetch

- 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

	iXX Dest	
Fetch	icode:ifun ← M₁[PC]	Read instruction byte
	valC ← M₈[PC+1]	Target address
	valP ← PC+5	Fall through address
Decode		
Execute	Cnd ← Cond(CC,ifun)	Take branch?
Memory		
Write back		
PC update	PC ← Cnd ? valC : valP	Update PC

- **Compute both addresses**

- **Choose based on setting of condition codes and branch condition**

Executing `call`

`call` Dest	8	0	Dest
return:	**XX**	**XX**	
target:	**XX**	**XX**	
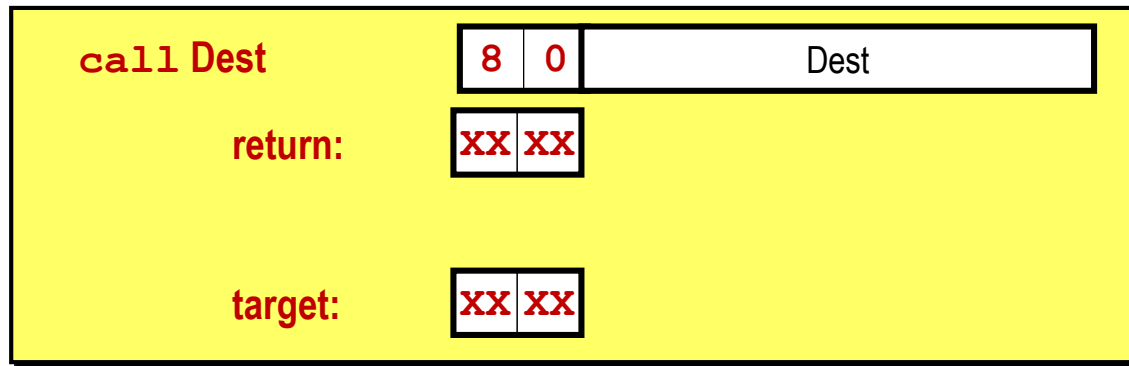
- **Fetch**
  - Read 5 bytes
  - Increment PC by 5
- **Decode**
  - Read stack pointer
- **Execute**
  - Decrement stack pointer by 4

- **Memory**
  - Write incremented PC to new value of stack pointer
- **Write back**
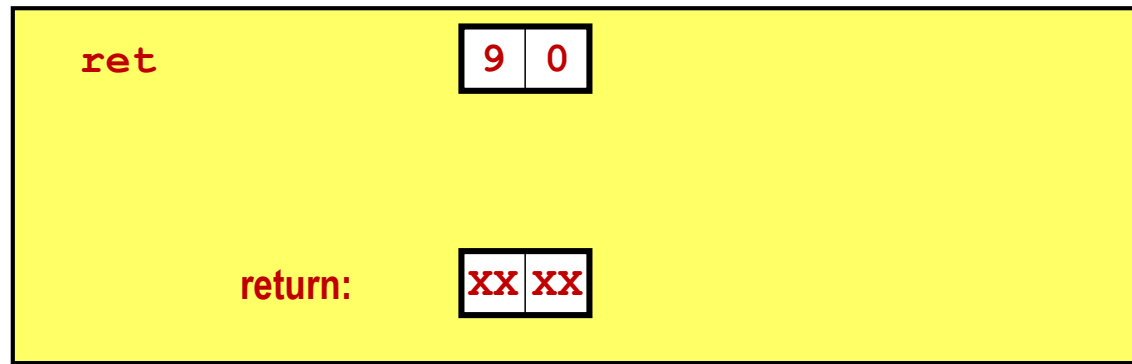 - Update stack pointer
- **PC Update**
  - Set PC to Dest

Stage Computation: `call`

	call Dest	
Fetch	icode:ifun ← M₁[PC]	Read instruction byte
	valC ← M₄[PC+1]	Target address
	valP ← PC+5	Return address
Decode	valB ← R[%esp]	Read stack pointer
Execute	valE ← valB + –4	Decrement stack pointer
Memory	M₄[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`

ret		**9** **0**
return:		**XX** **XX**

■**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$_1$[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 ← M₁[PC]	Read instruction byte
	rA,rB		[Read register byte]
	valC	valC ← M₄[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	M₄[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/Writeback**

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**

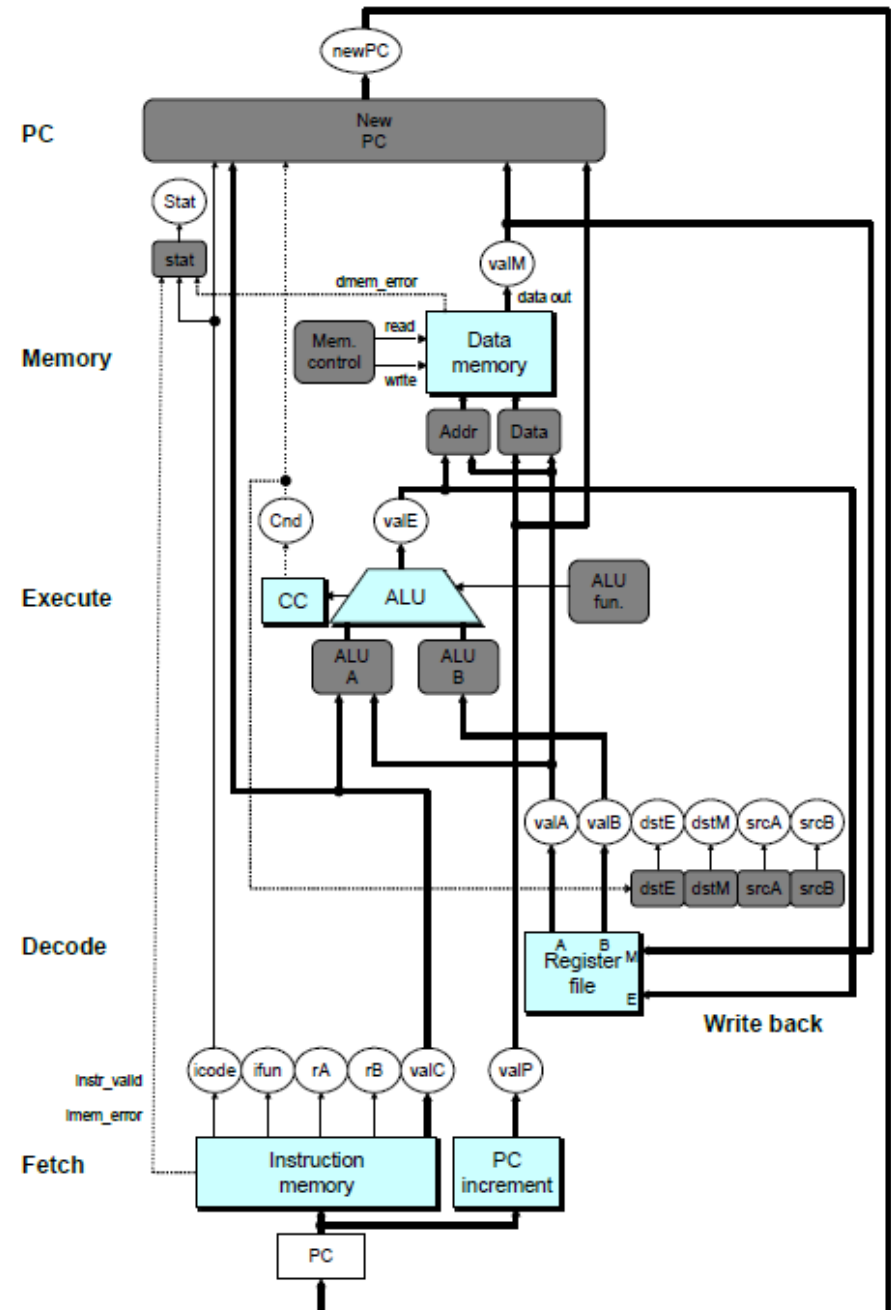| valE	ALU result
Cnd	Branch/move flag

■**Memory**

valM	Value from memory

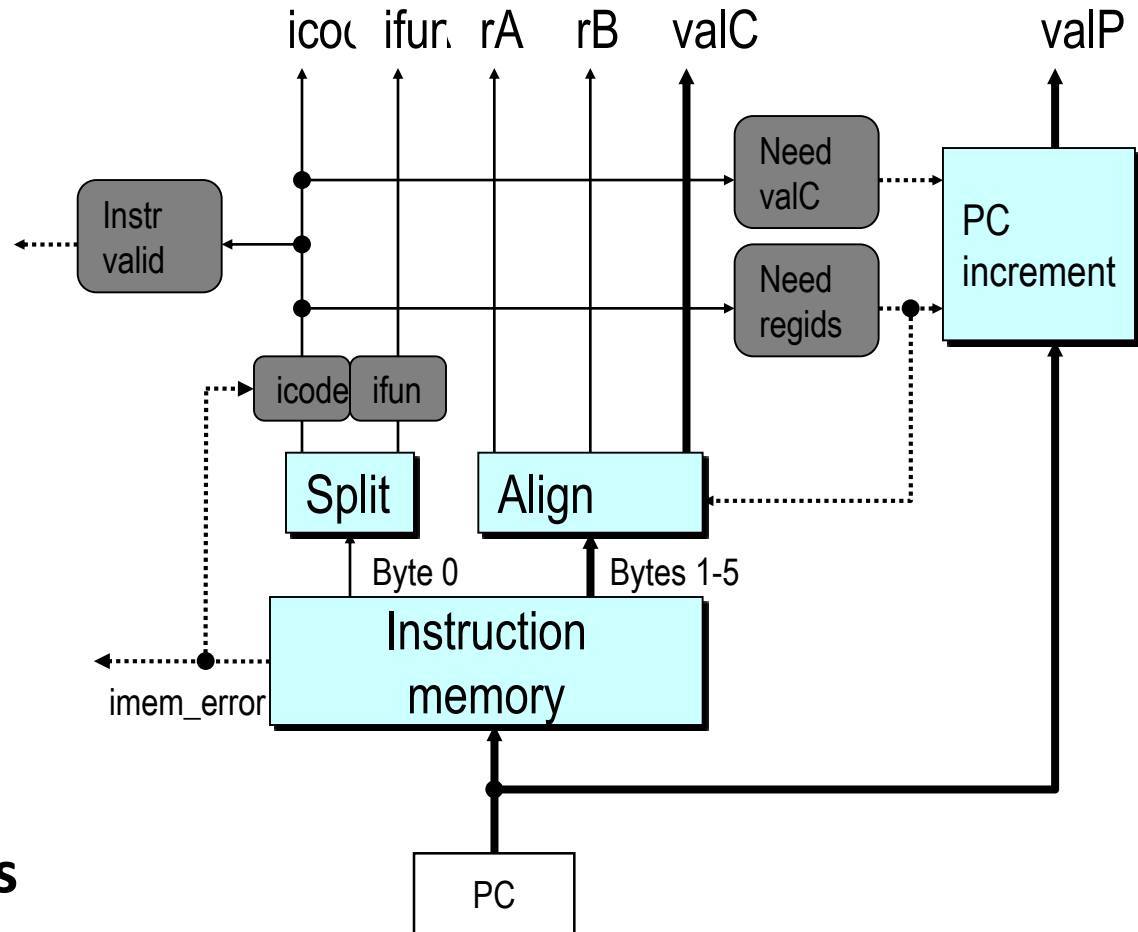# SEQ Hardware

- **Key**
  - Blue boxes:
    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

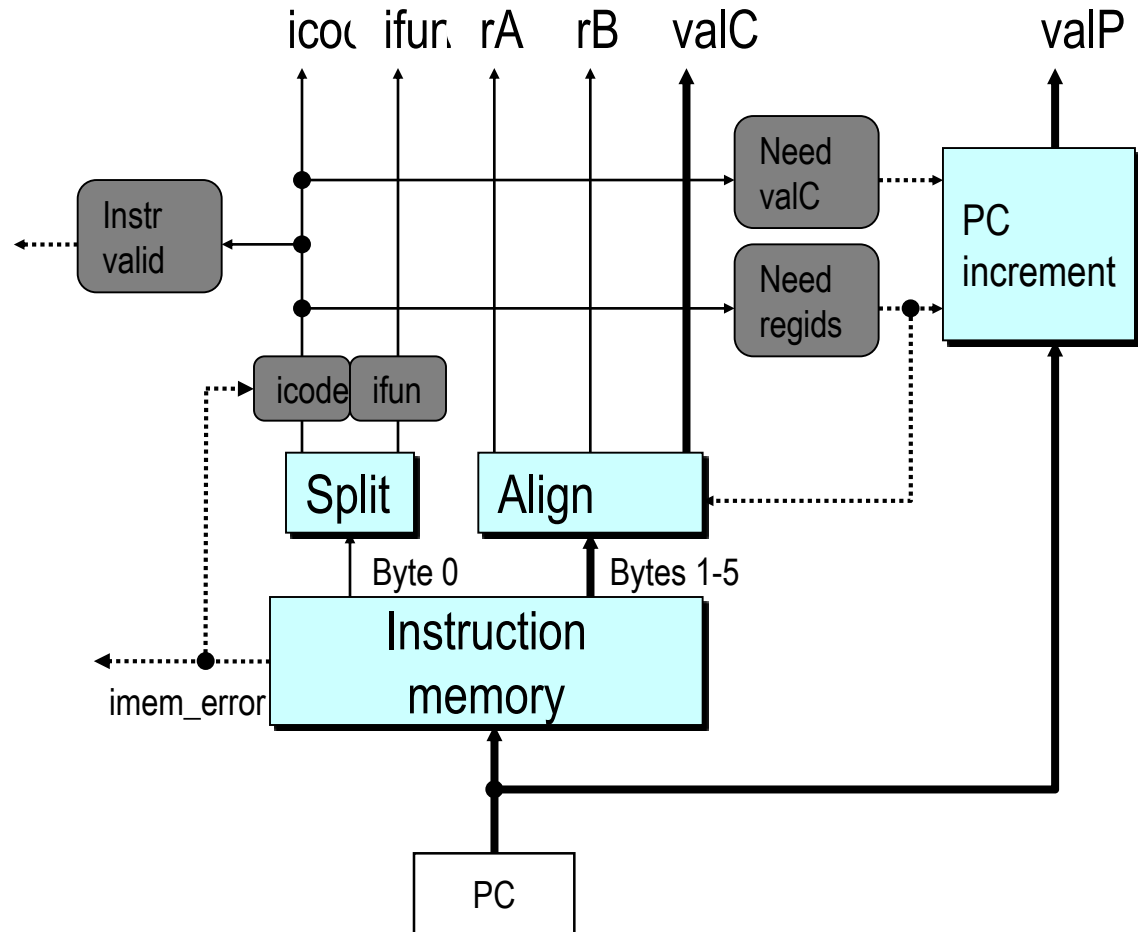# Fetch Logic

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

Fetch Logic

icode ifun rA   rB   valC   valP

Need valC

Instr valid

PC increment

Need regids

icode ifun

Split   Align

Byte 0   Bytes 1-5

Instruction memory

imem_error

PC

- **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?
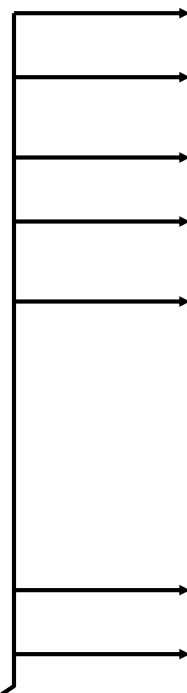
Fetch Control Logic in HCL

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

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

icode ifun

Split

Byte 0

Instruction memory

imem_error

PC
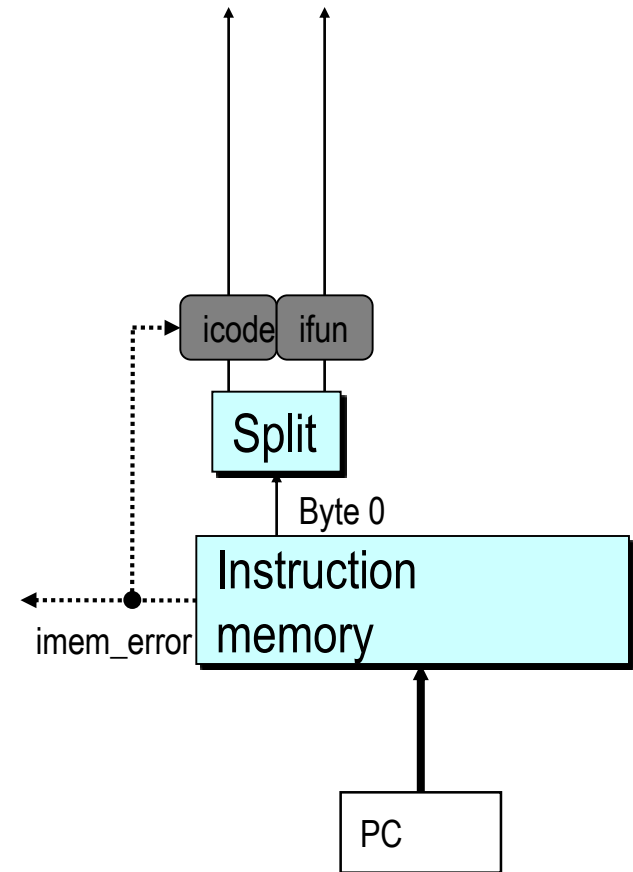
Fetch Control Logic in HCL

halt	0	0		

nop	1	0		

cmovXX rA, rB | 2 | fn | rA | rB |

irmovl V, rB | 3 | 0 | F | rB | V |

rmmovl rA, D(rB) | 4 | 0 | rA | rB | D |

mrmovl D(rB), rA | 5 | 0 | rA | rB | D |

OPl rA, rB | 6 | fn | rA | rB |

jXX Dest | 7 | fn | Dest |

call Dest | 8 | 0 | Dest |

ret | 9 | 0 |

pushl rA | A | 0 | rA | F |

popl rA | B | 0 | rA | F |

```
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 };
```
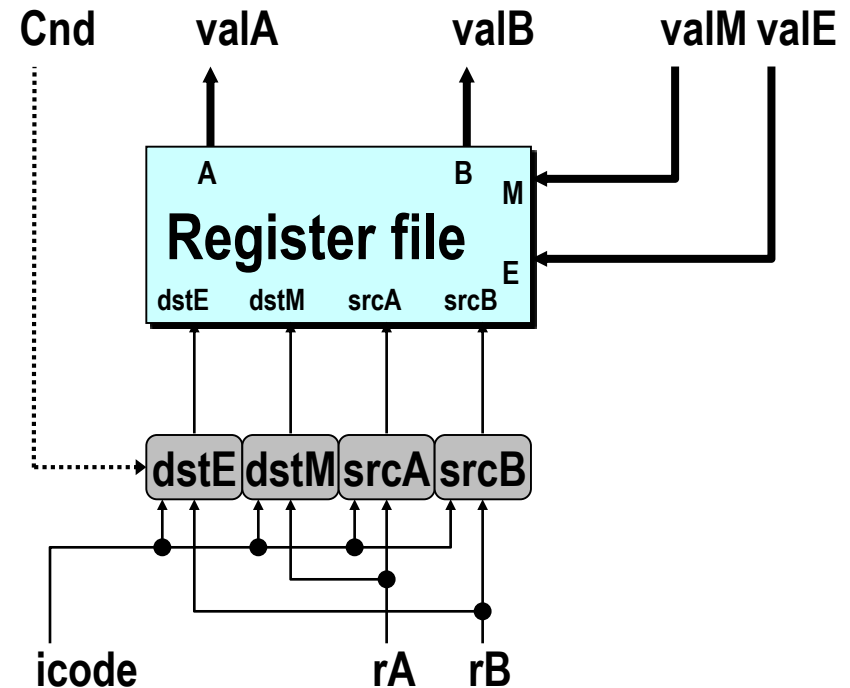
Decode Logic

- ## Register File
    - Read ports A, B
    - Write ports E, M
    - Addresses are register IDs or 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

	OPl rA, rB	
Decode	valA ← R[rA]	Read operand A

	cmovXX rA, rB	
Decode	valA ← R[rA]	Read operand A

	`rmmovl` rA, D(rB)	
Decode	valA ← R[rA]	Read operand A

	`popl` rA	
Decode	valA ← R[%esp]	Read stack pointer

	jXX Dest	
Decode		No operand

	`call` Dest	
Decode		No operand

	`ret`	
Decode	valA ← R[%esp]	Read stack pointer

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

27

E Desti-
nation

	OPl rA, rB
Write-back	R[**rB**] ← valE

Write back result

	cmovXX rA, rB
Write-back	R[**rB**] ← valE

Conditionally write back result

	`rmmovl` rA, D(rB)
Write-back	

None

	`popl` rA
Write-back	R[**%esp**] ← valE

Update stack pointer

	jXX Dest
Write-back	

None

	`call` Dest
Write-back	R[**%esp**] ← valE

Update stack pointer

	`ret`
Write-back	R[**%esp**] ← valE

Update stack pointer

```
int dstE = [
      icode in { IRRMOVL } && Cnd : rB;
      icode in { IIRMOVL, IOPL} : rB;
      icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
      1 : RNONE;  # Don't write any register
];
```
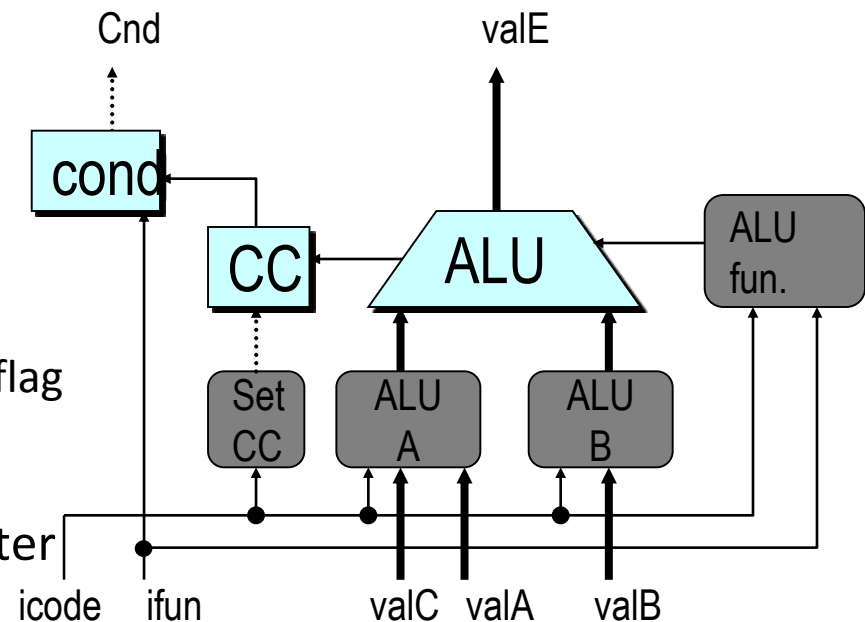
28

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

Execute	OPl rA, rB
	valE ← valB OP valA

Perform ALU operation

Execute	cmovXX rA, rB
	valE ← 0 + valA

Pass valA through ALU

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

Compute effective address

Execute	`popl` rA
	valE ← valB + 4

Increment stack pointer

Execute	jXX Dest

No operation

Execute	`call` Dest
	valE ← valB + –4

Decrement stack pointer

Execute	`ret`
	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 Oper-ation

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

`cmovXX` rA, rB		
Execute	valE ← 0 **+** valA	**Pass valA through ALU**

`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;
];
```
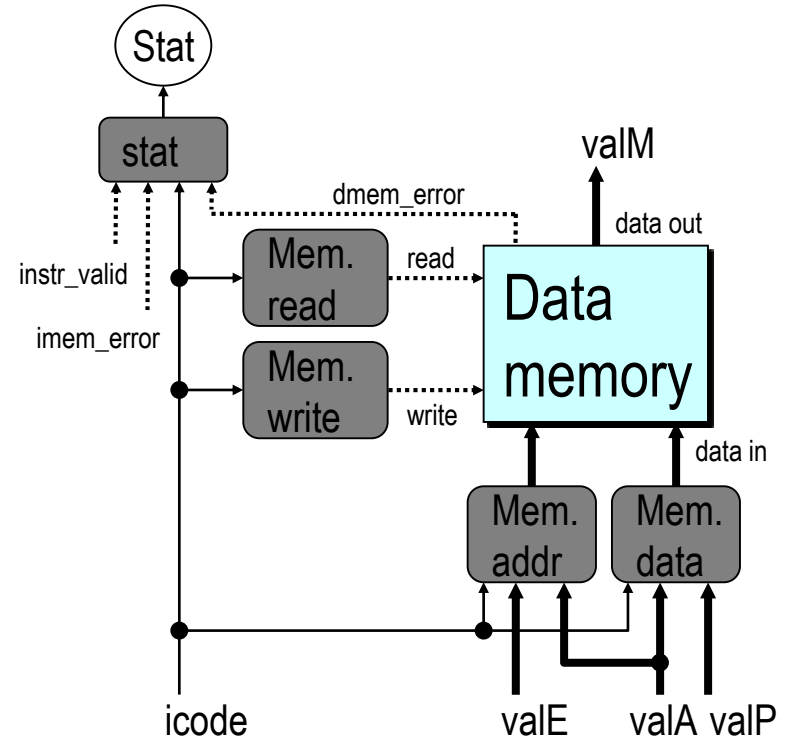
Memory Logic

- **Memory**
  - Reads or writes memory word
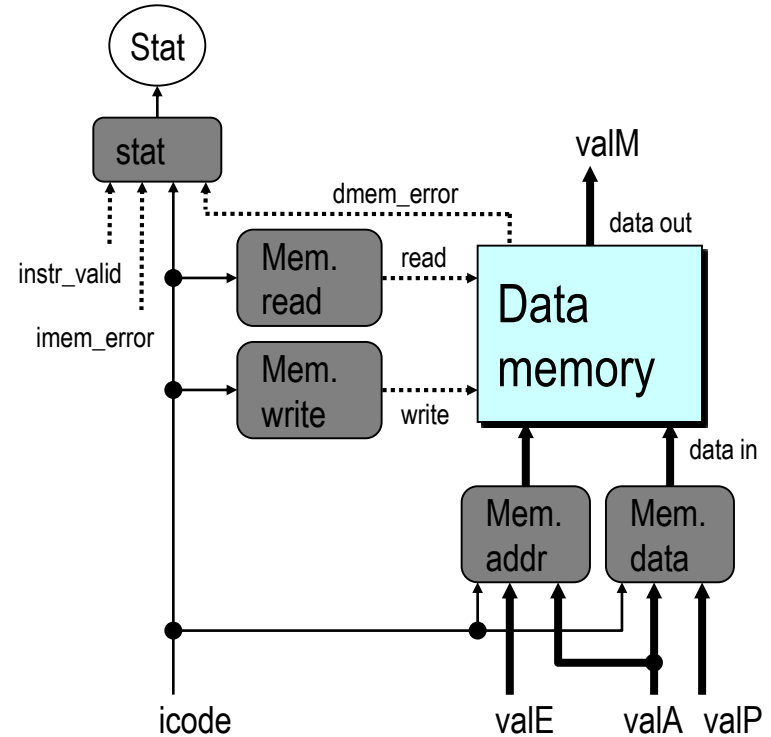- **Control Logic**
  - stat: What is instruction status?
  - Mem. read: should word be read?
  - Mem. write: should word be written?
  - Mem. addr.: Select address
  - Mem. data.: Select data

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;
];
```

33

Memory Address

OPl rA, rB		
Memory		**No operation**

rmmovl rA, D(rB)		
Memory	$M_4[valE] \leftarrow valA$	**Write value to memory**

popl rA		
Memory	$valM \leftarrow M_4[valA]$	**Read from stack**

jXX Dest		
Memory		**No operation**

call Dest		
Memory	$M_4[valE] \leftarrow valP$	**Write return value on stack**

ret		
Memory	$valM \leftarrow M_4[valA]$	**Read return address**

```
int mem_addr = [
      icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
      icode in { IPOPL, IRET } : valA;
      # Other instructions don't need address
];
```
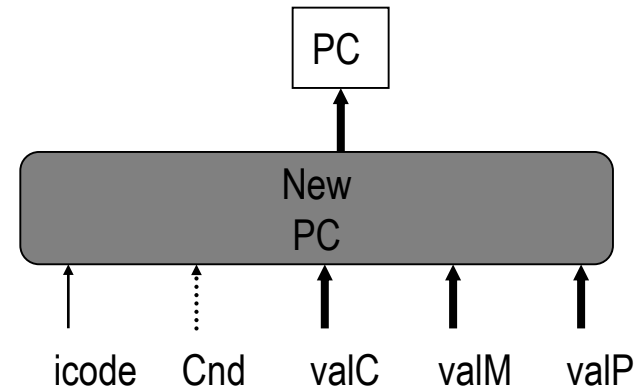
34

Memory Read

OPl rA, rB	
Memory	

No operation

`rmmovl` rA, D(rB)	
Memory	$M_4[valE] \leftarrow valA$

Write value to memory

`popl` rA	
Memory	$valM \leftarrow M_4[valA]$

Read from stack

jXX Dest	
Memory	

No operation

`call` Dest	
Memory	$M_4[valE] \leftarrow valP$

Write return value on stack

`ret`	
Memory	$valM \leftarrow M_4[valA]$

Read return address

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

PC Update Logic

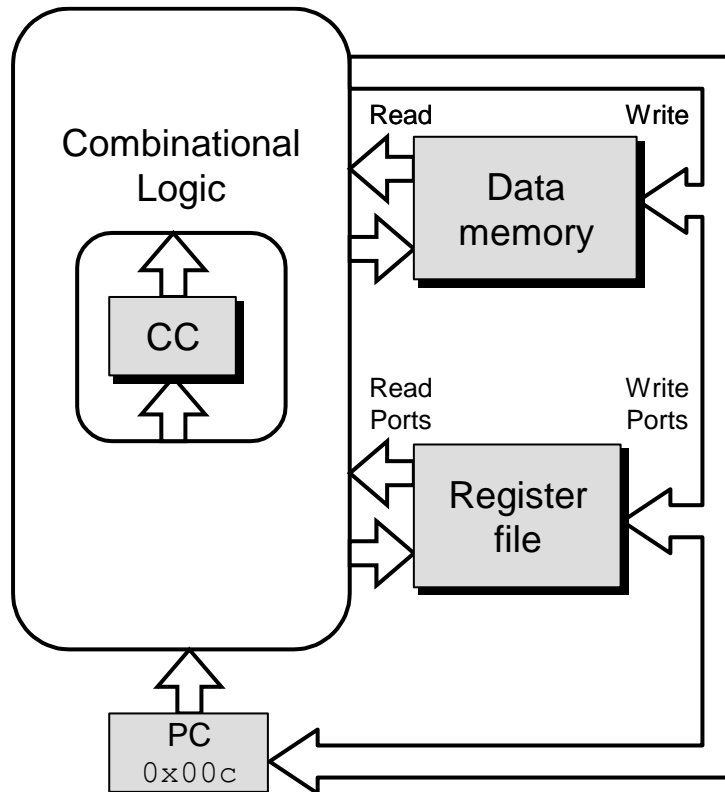- **New PC**
  - Select next value of PC

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 ← Cnd ? 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 && Cnd : valC;
        icode == IRET : valM;
        1 : valP;
];
```

37

SEQ Operation

- **State**
  - PC register
  - Cond. Code register
  - Data memory
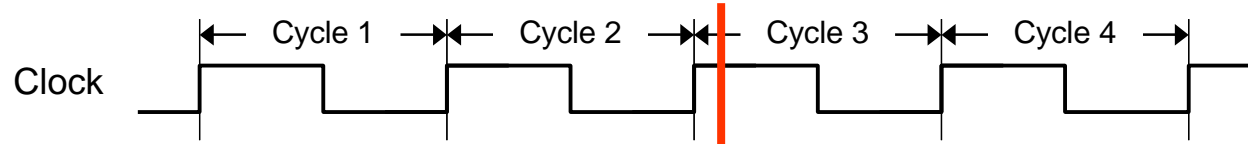 - Register file
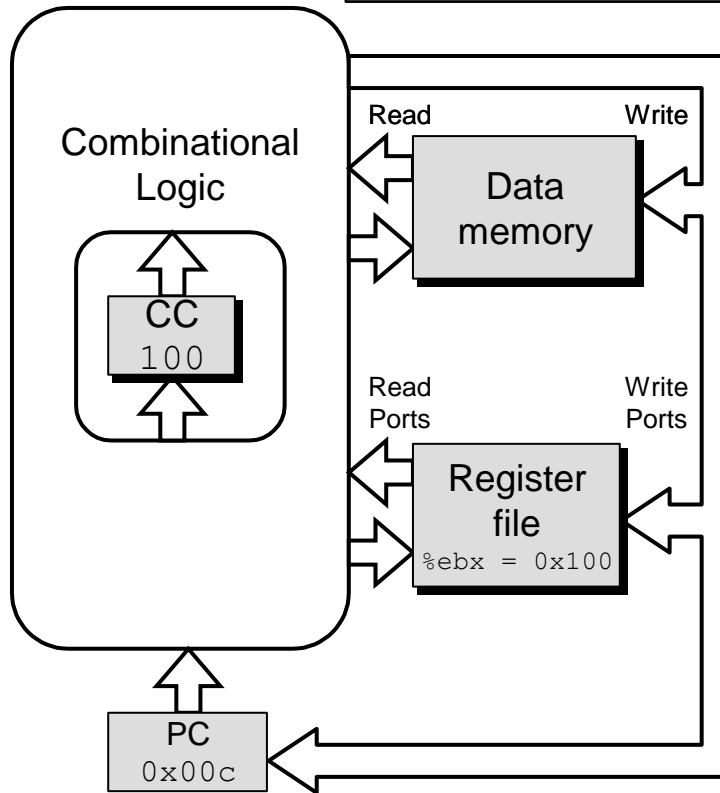
  All updated as clock rises

- **Combinational Logic**
  - ALU
  - Control logic
  - Memory reads
    - Instruction memory
   - Register file
    - Data memory

SEQ Operation #2

Clock | Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4

```
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
```

Combinational Logic

Read   Write
Data memory

CC
100

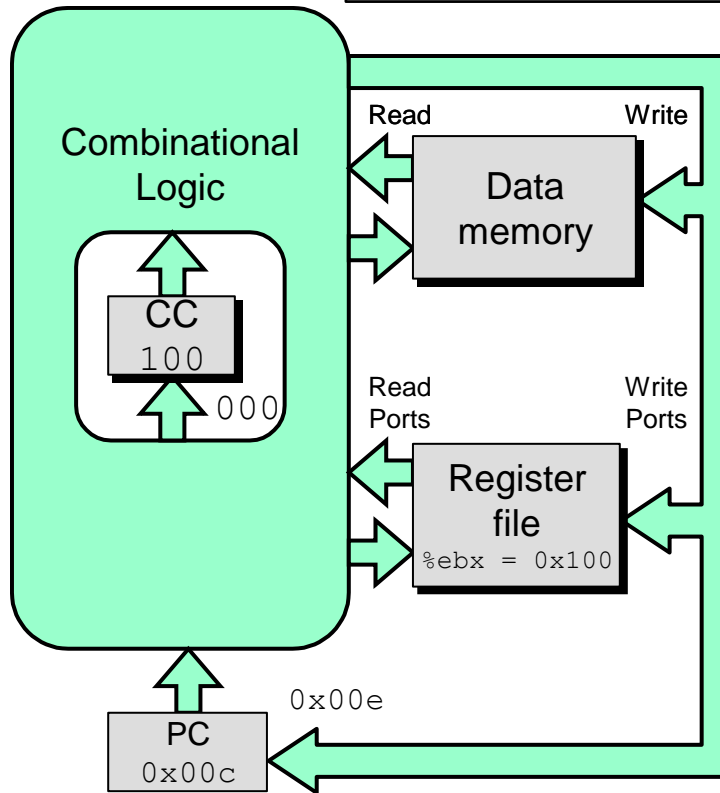Read Ports   Write Ports
Register file
%ebx = 0x100

PC
0x00c

- **state set according to second `irmovl` instruction**

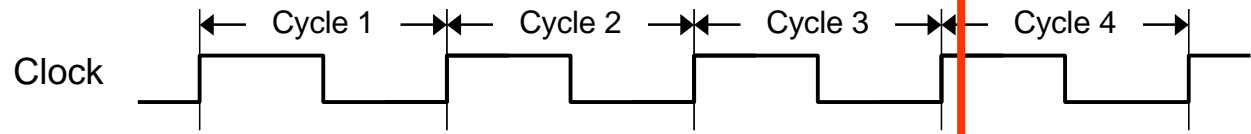- **combinational logic starting to react to state changes**

SEQ Operation #3
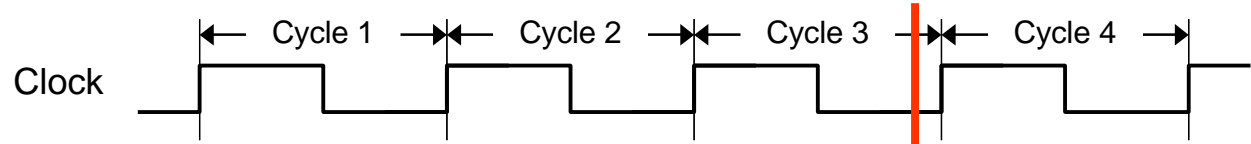
Cycle 1 → ← Cycle 2 → ← Cycle 3 → ← Cycle 4 →

Clock

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`

Combinational Logic

Read   Write

Data memory

CC 100

000

Read Ports   Write Ports
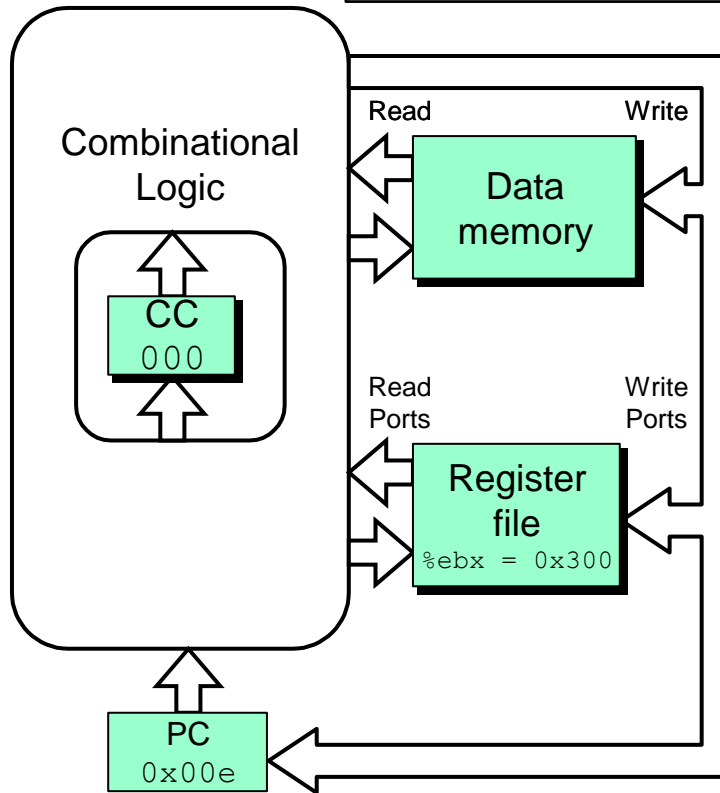
Register file

`%ebx = 0x100`

`0x00e`

PC `0x00c`

- **state set according to second `irmovl` instruction**

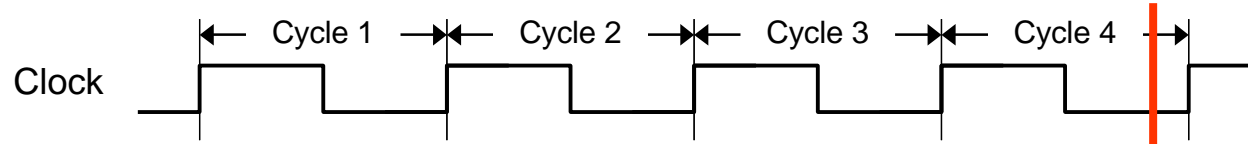- **combinational logic generates results for `addl` instruction**

40

SEQ Operation #4

Clock

	Cycle 1	Cycle 2	Cycle 3	Cycle 4

```
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
```

Combinational Logic

Read   Write

Data memory

CC
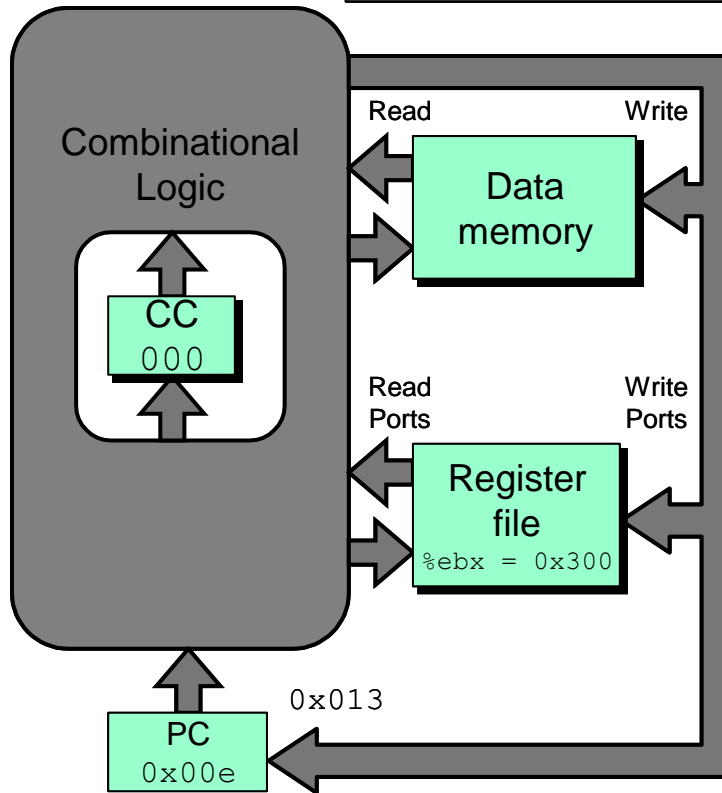000

Read Ports   Write Ports

Register file
%ebx = 0x300

PC
0x00e

- **state set according to `addl` instruction**

- **combinational logic starting to react to state changes**

41

SEQ Operation #5

Clock

	Cycle 1	Cycle 2	Cycle 3	Cycle 4

```
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
```

Combinational Logic

Read   Write

Data memory

CC
000

Read Ports   Write Ports

Register file
%ebx = 0x300

0x013

PC
0x00e

- **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
- **Pros**
  - Simple but complete
  - Physically realizable
- **Cons**
  - 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