

# Processor Architecture II: SEQ: Sequential Implementation

Introduction to Computer Systems  
10<sup>th</sup> Lecture, Oct 18, 2017

## Instructors:

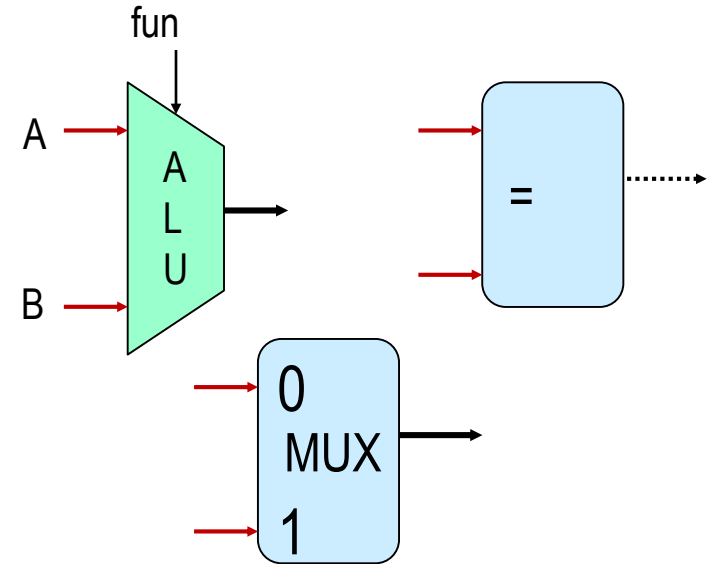
Xiangqun Chen , Junlin Lu

Guangyu Sun , Xuetao Guan

# Building Blocks

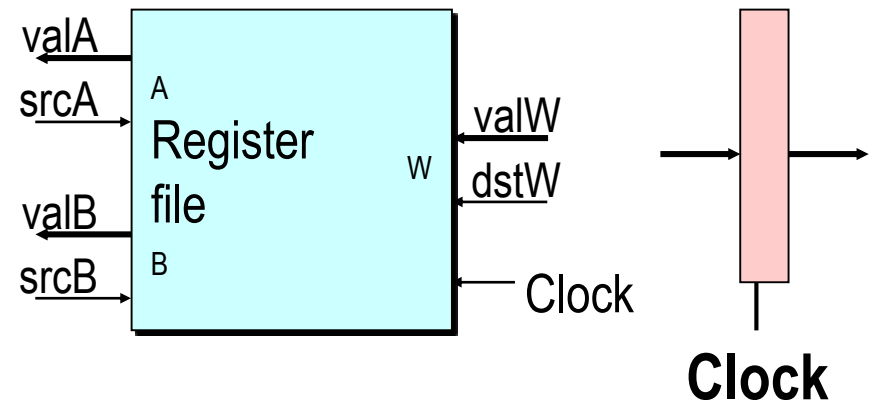
## ■ Combinational Logic

- 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



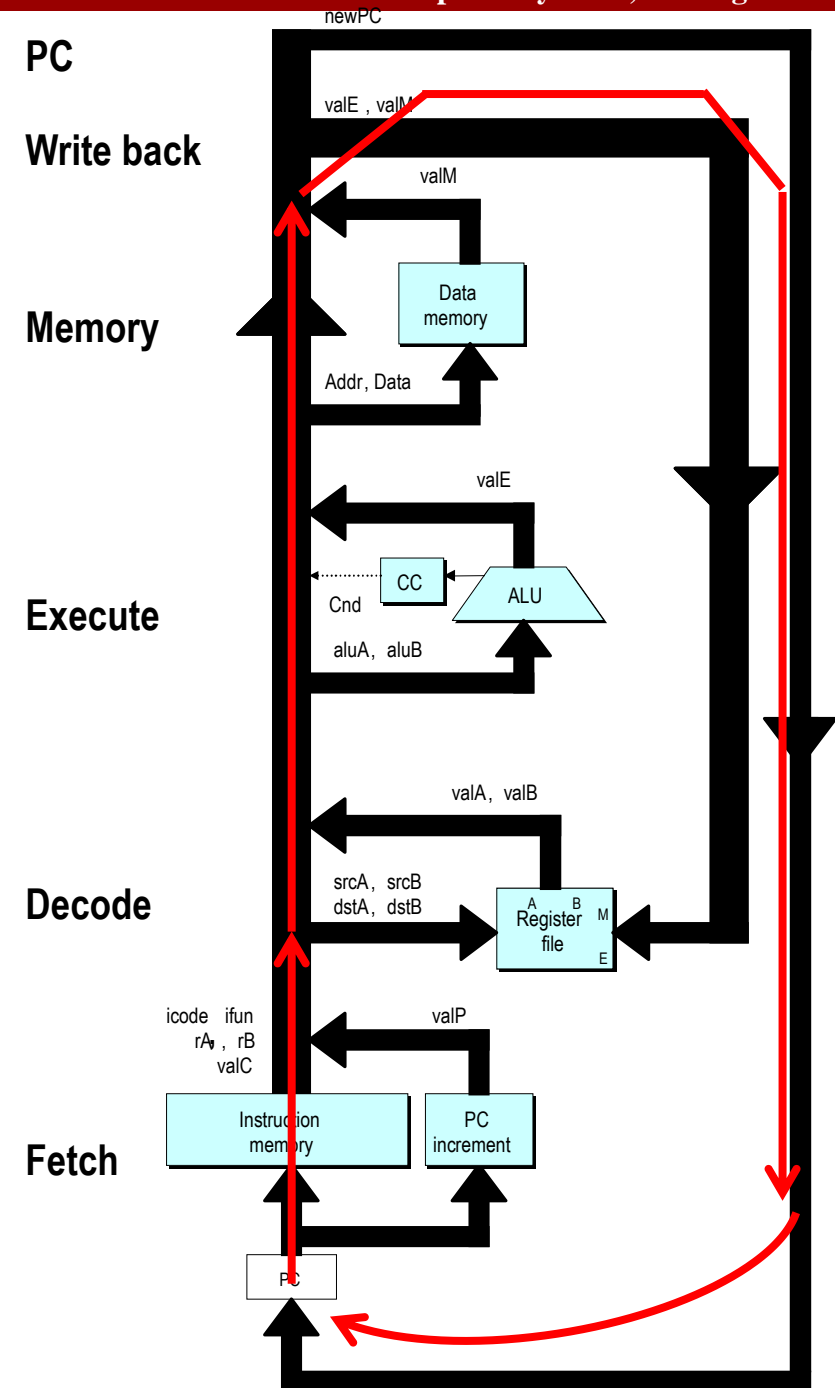
# SEQ Hardware Structure

## ■ 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

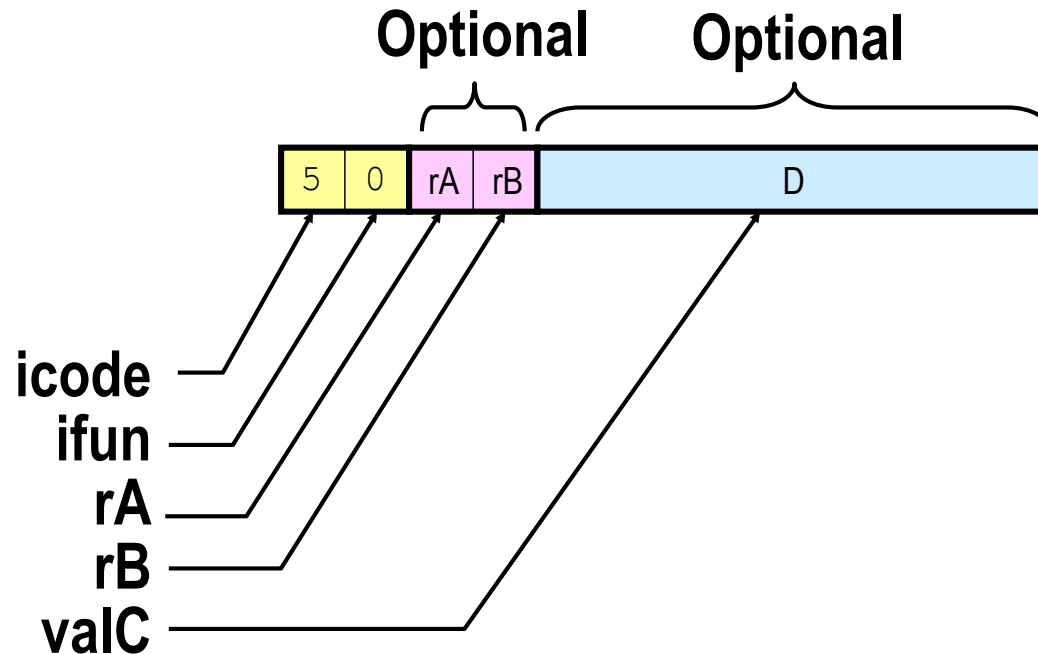
## ■ Instruction Flow

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





# Instruction Decoding



## ■ Instruction Format

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

# Executing Arith./Logical Operation



## ■ 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

	OPI rA, rB	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$	Read instruction byte Read register byte
	$\text{valP} \leftarrow \text{PC}+2$	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 `rmmovl` (store)



## ■ 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

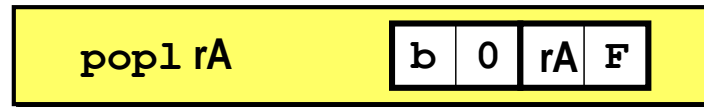


# Stage Computation: `rmmovl`

	<code>rmmovl rA, D(rB)</code>	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valC} \leftarrow M_4[\text{PC}+2]$ $\text{valP} \leftarrow \text{PC}+6$	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	$M_4[\text{valE}] \leftarrow \text{valA}$	Write value to memory
Write back		
PC update	$\text{PC} \leftarrow \text{valP}$	Update PC

- Use ALU for address computation

# Executing 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

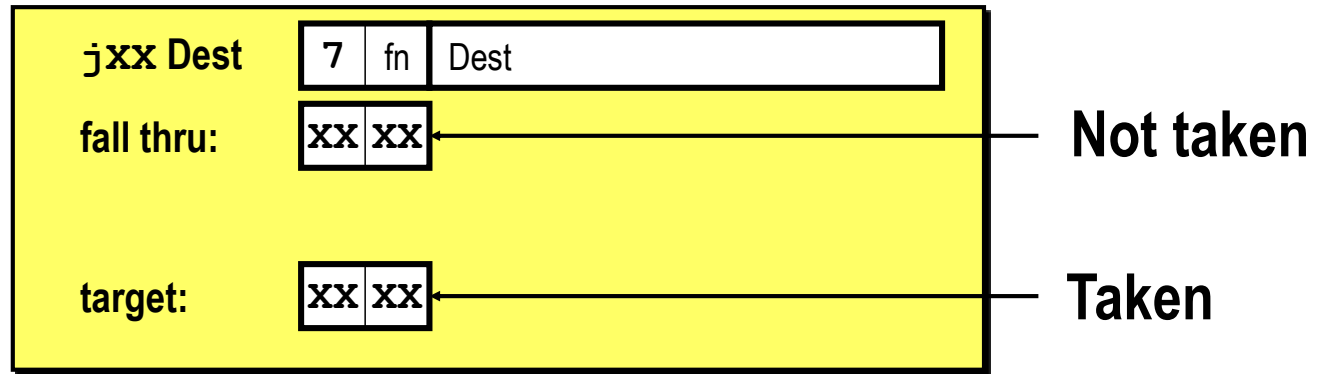
- Increment PC by 2

# Stage Computation: popl

	popl rA	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$	Read instruction byte Read register byte
	$\text{valP} \leftarrow \text{PC}+2$	Compute next PC
Decode	$\text{valA} \leftarrow R[\%esp]$ $\text{valB} \leftarrow R[\%esp]$	Read stack pointer Read stack pointer
Execute	$\text{valE} \leftarrow \text{valB} + 4$	Increment stack pointer
Memory	$\text{valM} \leftarrow M_1[\text{valA}]$	Read from stack
Write back	$R[\%esp] \leftarrow \text{valE}$ $R[\text{rA}] \leftarrow \text{valM}$	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 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

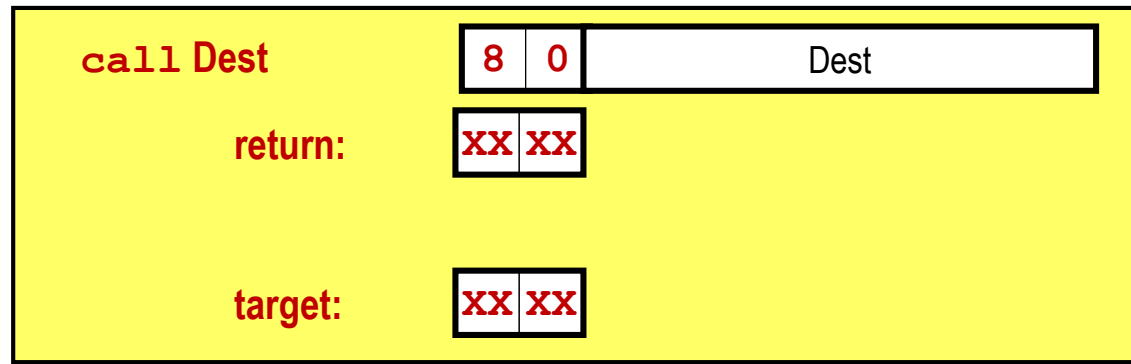
- Set PC to Dest if branch taken or to incremented PC if not branch

# Stage Computation: Jumps

	iXX Dest	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_4[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+5$	Read instruction byte Target address Fall through address
Decode		
Execute	$\text{Cnd} \leftarrow \text{Cond}(\text{CC.ifun})$	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 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

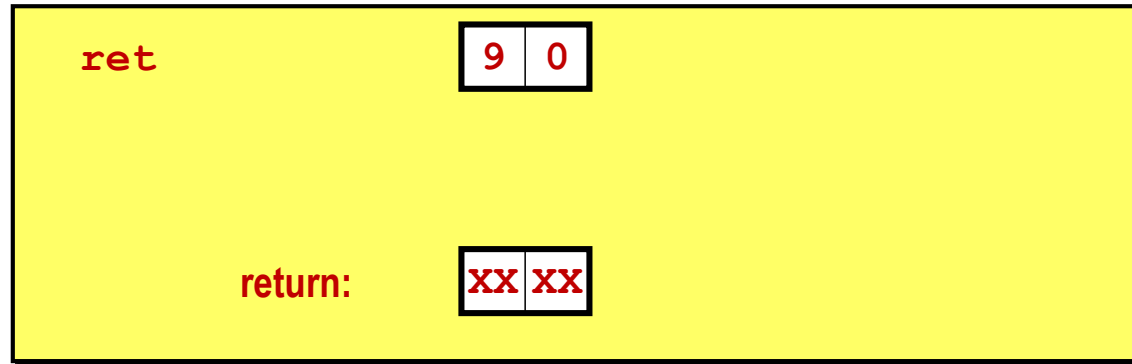
- Set PC to Dest

# Stage Computation: `call`

	<code>call</code> Dest	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+5$	Read instruction byte Target address Return address
Decode	$\text{valB} \leftarrow R[\%esp]$	Read stack pointer
Execute	$\text{valE} \leftarrow \text{valB} + -4$	Decrement stack pointer
Memory	$M_4[\text{valE}] \leftarrow \text{valP}$	Write return value on stack
Write back	$R[\%esp] \leftarrow \text{valE}$	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 4

## ■ 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	$\text{valA} \leftarrow R[\%esp]$ $\text{valB} \leftarrow R[\%esp]$	Read operand stack pointer Read operand stack pointer
Execute	$\text{valE} \leftarrow \text{valB} + 4$	Increment stack pointer
Memory	$\text{valM} \leftarrow M_4[\text{valA}]$	Read return address
Write back	$R[\%esp] \leftarrow \text{valE}$	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

		OPI 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 $\leftarrow valB \text{ OP } valA$
	Cond code	Set CC
Memory	valM	
Write back	dstE	R[rB] $\leftarrow valE$
	dstM	
PC update	PC	PC $\leftarrow valP$

Read instruction byte  
 Read register byte  
 [Read constant word]  
 Compute next PC  
 Read operand A  
 Read operand B  
 Perform ALU operation  
 Set condition code register  
 [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	icode:ifun $\leftarrow M_1[PC]$
	rA,rB	
	valC	valC $\leftarrow M_4[PC+1]$
	valP	valP $\leftarrow PC+5$
Decode	valA, srcA	
	valB, srcB	valB $\leftarrow R[\%esp]$
Execute	valE	valE $\leftarrow valB + -4$
	Cond code	
Memory	valM	$M_4[valE] \leftarrow valP$
Write back	dstE	$R[\%esp] \leftarrow valE$
	dstM	
PC update	PC	PC $\leftarrow valC$

Read instruction byte  
 [Read register byte]  
 Read constant word  
 Compute next PC  
 [Read operand A]  
 Read operand B  
 Perform ALU operation  
 [Set condition 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/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

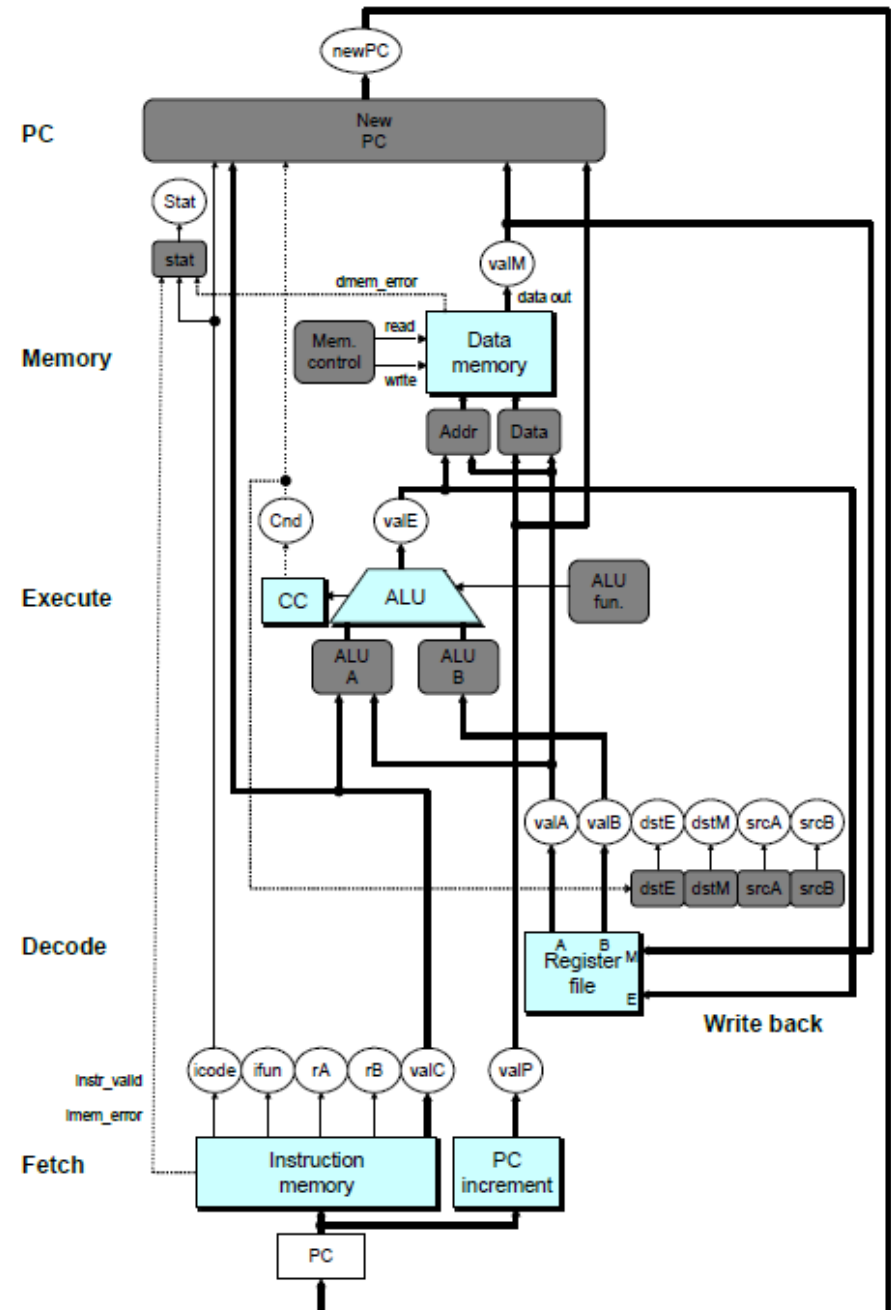
## ■ Memory

- valM    Value from memory

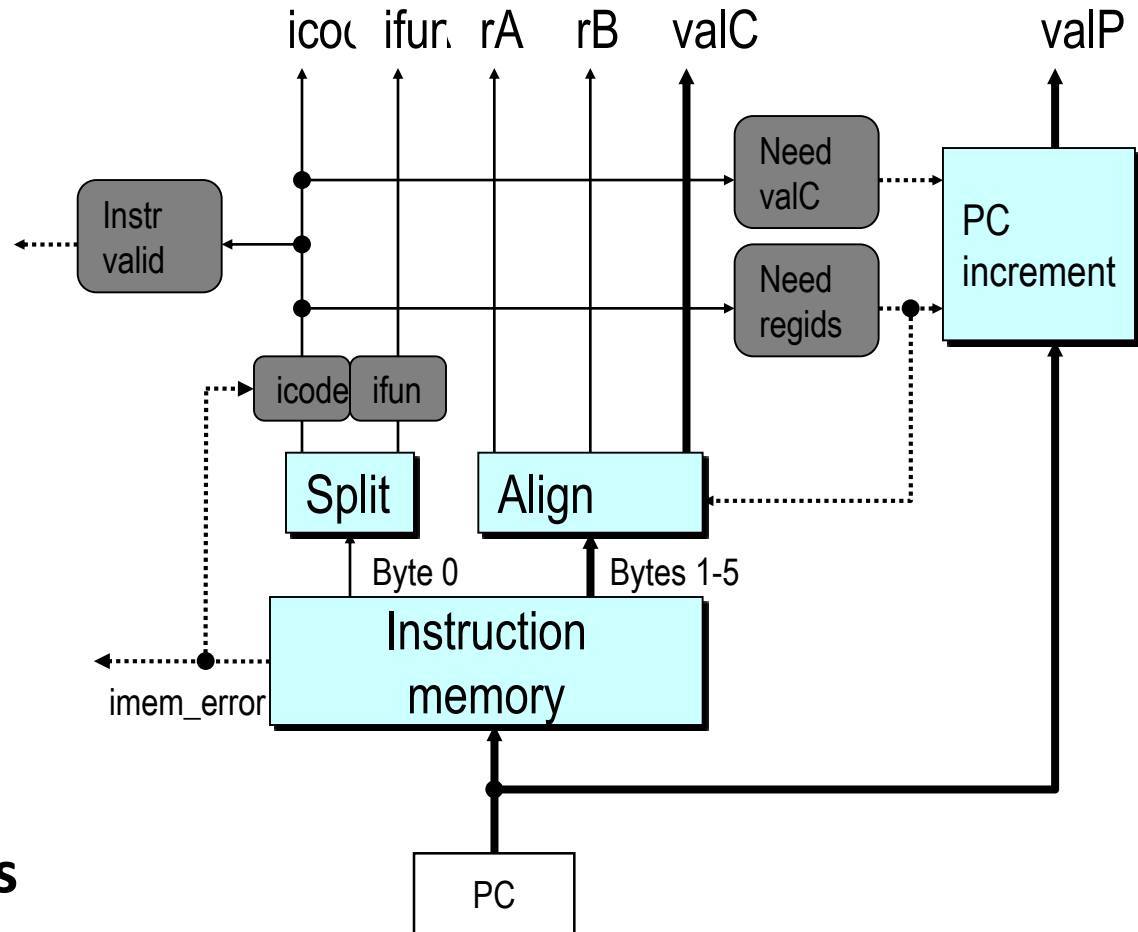
# SEQ Hardware

## ■ 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



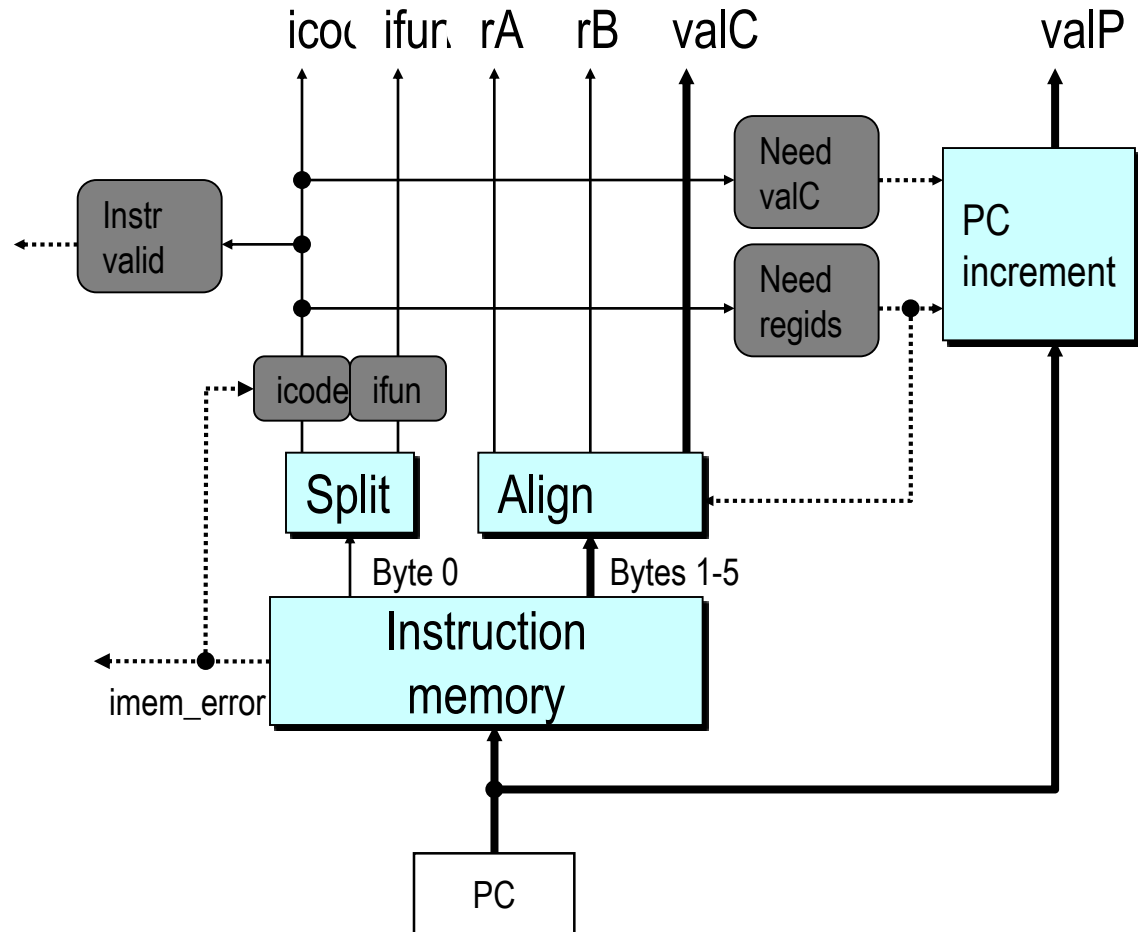
# Fetch Logic



## ■ 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



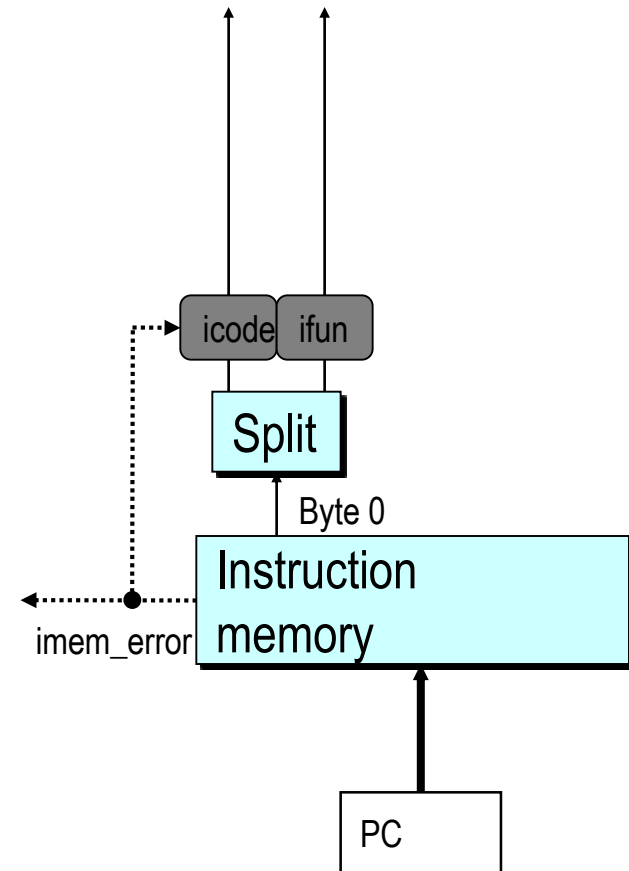
## ■ Control Logic

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

# Fetch Control Logic in HCL

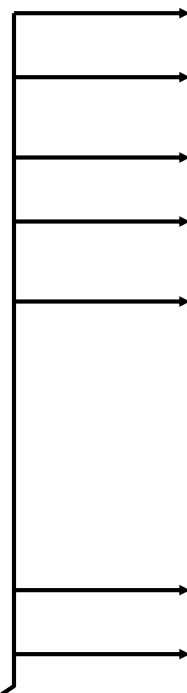
```
# Determine instruction code
int icode = [
    imem_error: INOP;
    1: imem_icode;
];

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





# Fetch Control Logic in HCL



halt	0	0		
nop	1	0		
cmovXX rA, rB	2	fn	rA	rB
irmovl V, rB	3	0	F	rB
rmmovl rA, D(rB)	4	0	rA	rB
mrmmovl D(rB), rA	5	0	rA	rB
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

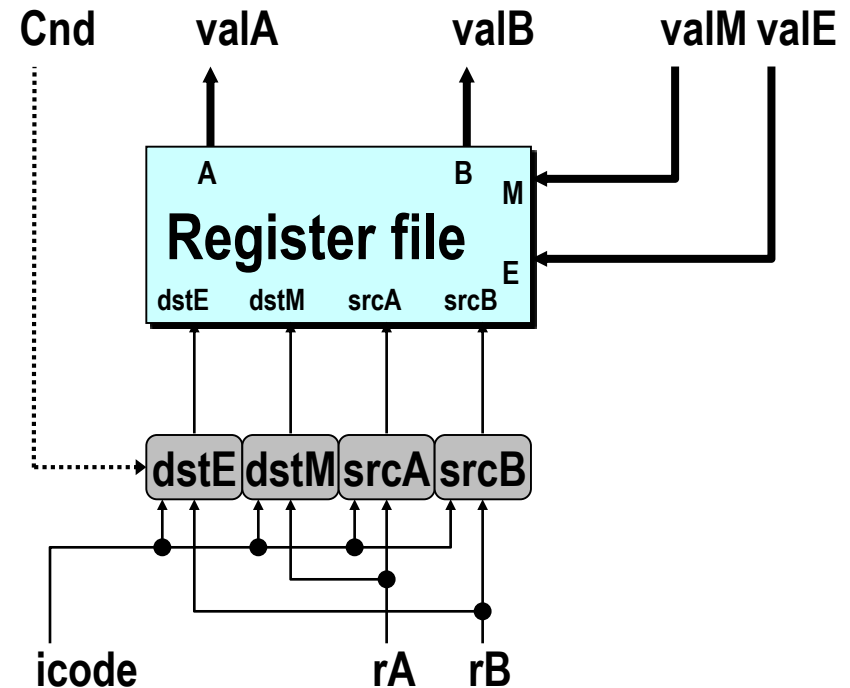
- Read ports A, B
- Write ports E, M
- Addresses are register IDs or 15 (0xF) (no access)

## 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	$valA \leftarrow R[rA]$	Read operand A
	cmovXX rA, rB	
Decode	$valA \leftarrow R[rA]$	Read operand A
	rmmovl rA, D(rB)	
Decode	$valA \leftarrow R[rA]$	Read operand A
	popl rA	
Decode	$valA \leftarrow R[\%esp]$	Read stack pointer
	jXX Dest	
Decode		No operand
	call Dest	
Decode		No operand
	ret	
Decode	$valA \leftarrow R[\%esp]$	Read stack pointer

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

# E Destination

	OPl rA, rB	
Write-back	$R[rB] \leftarrow valE$	Write back result
	cmovXX rA, rB	
Write-back	$R[rB] \leftarrow valE$	Conditionally write back result
	rmmovl rA, D(rB)	
Write-back		None
	popl rA	
Write-back	$R[\%esp] \leftarrow valE$	Update stack pointer
	jXX Dest	
Write-back		None
	call Dest	
Write-back	$R[\%esp] \leftarrow valE$	Update stack pointer
	ret	
Write-back	$R[\%esp] \leftarrow valE$	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
];

```

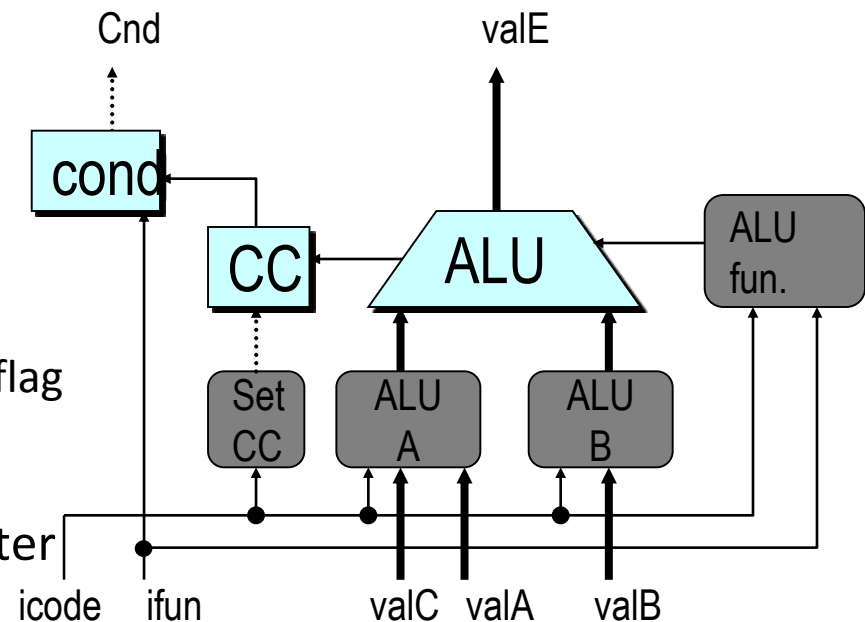
# Execute Logic

## ■ 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

	OPl 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
	popl rA	
Execute	$\text{valE} \leftarrow \text{valB} + 4$	Increment stack pointer
	jXX Dest	
Execute		No operation
	call Dest	
Execute	$\text{valE} \leftarrow \text{valB} + -4$	Decrement stack pointer
	ret	
Execute	$\text{valE} \leftarrow \text{valB} + 4$	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	$\text{valE} \leftarrow \text{valB} \text{ OP } \text{valA}$	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
	popl rA	
Execute	$\text{valE} \leftarrow \text{valB} + 4$	Increment stack pointer
	jXX Dest	
Execute		No operation
	call Dest	
Execute	$\text{valE} \leftarrow \text{valB} + -4$	Decrement stack pointer
	ret	
Execute	$\text{valE} \leftarrow \text{valB} + 4$	Increment stack pointer

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

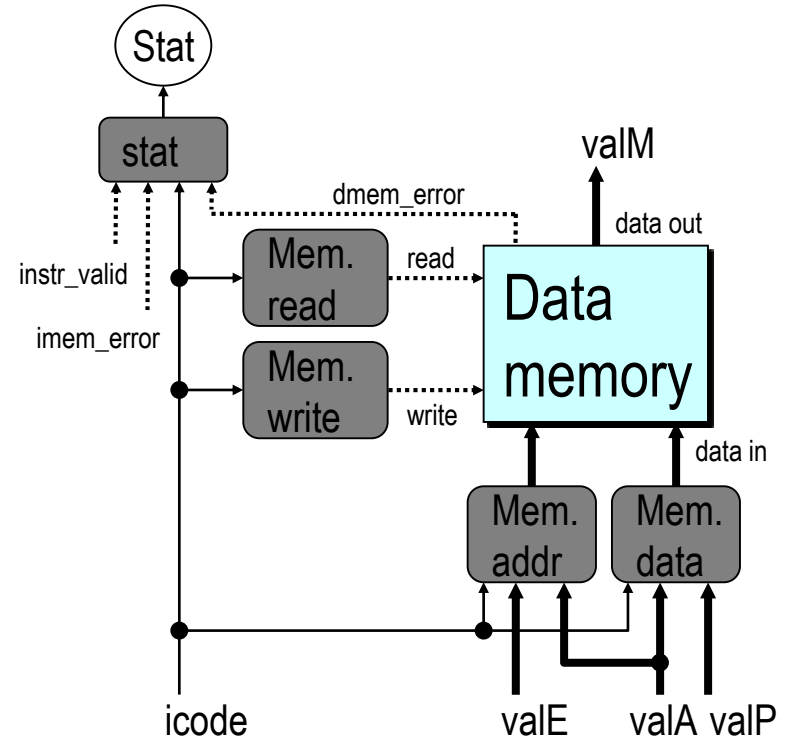
# Memory Logic

## ■ 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

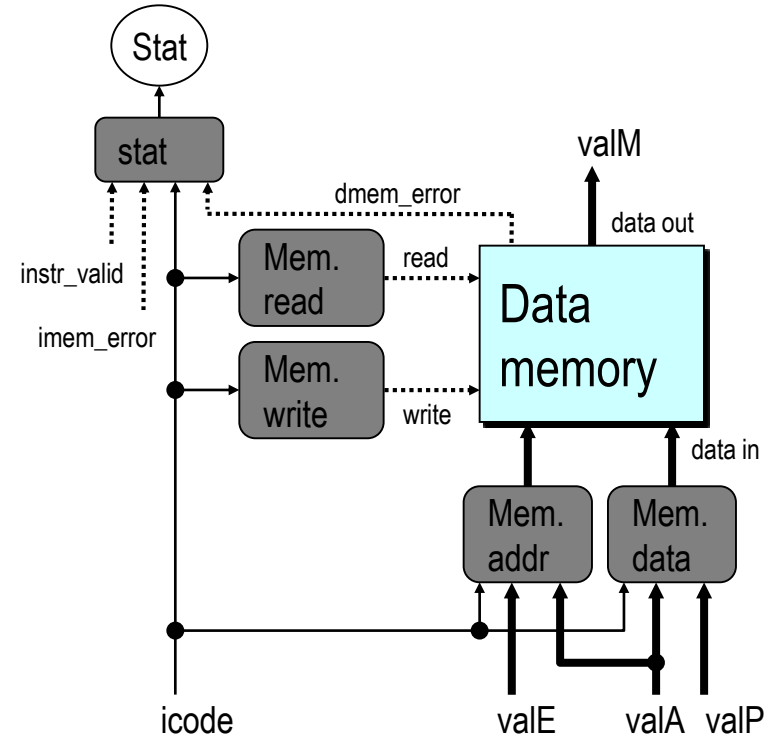




# Instruction Status

## ■ 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;
];
```

# Memory Address

	OPl rA, rB	
Memory		No operation
	rmmovl rA, D(rB)	
Memory	$M_4[\text{valE}] \leftarrow \text{valA}$	Write value to memory
	popl rA	
Memory	$\text{valM} \leftarrow M_4[\text{valA}]$	Read from stack
	jXX Dest	
Memory		No operation
	call Dest	
Memory	$M_4[\text{valE}] \leftarrow \text{valP}$	Write return value on stack
	ret	
Memory	$\text{valM} \leftarrow M_4[\text{valA}]$	Read return address

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

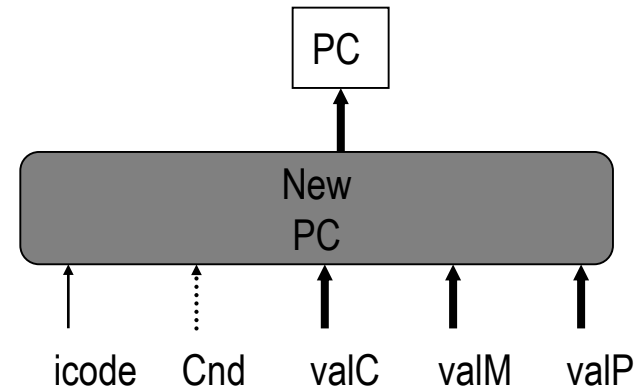
# Memory Read

	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

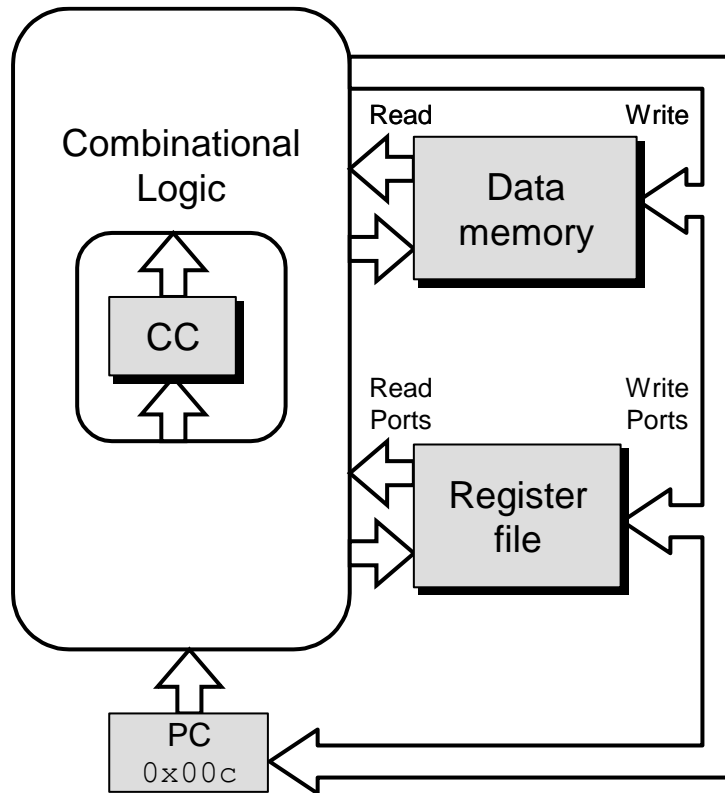


# PC Update

	OPl rA, rB	
PC update	PC $\leftarrow$ valP	Update PC
	rmmovl rA, D(rB)	
PC update	PC $\leftarrow$ valP	Update PC
	popl 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

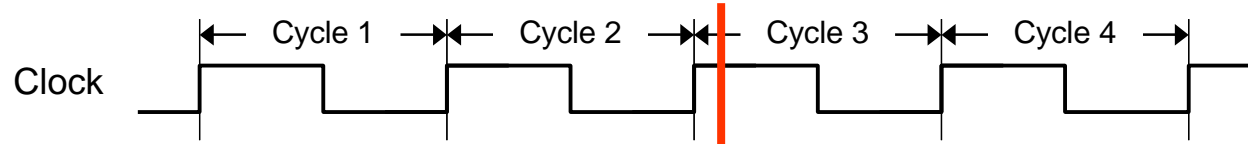
*All updated as clock rises*

## ■ Combinational Logic

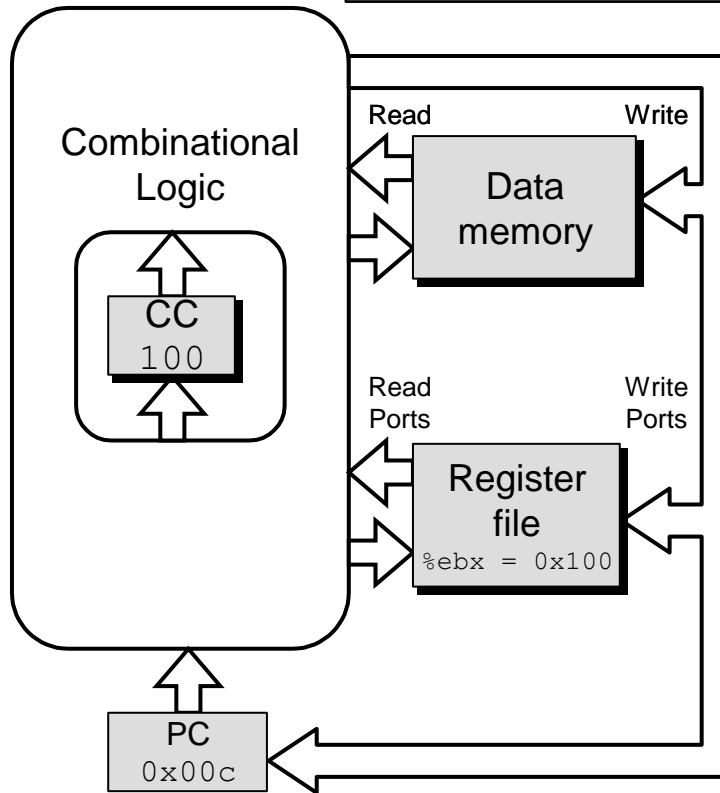
- ALU
- Control logic
- Memory reads
  - Instruction memory
  - Register file
  - Data memory

# SEQ

## Operation #2



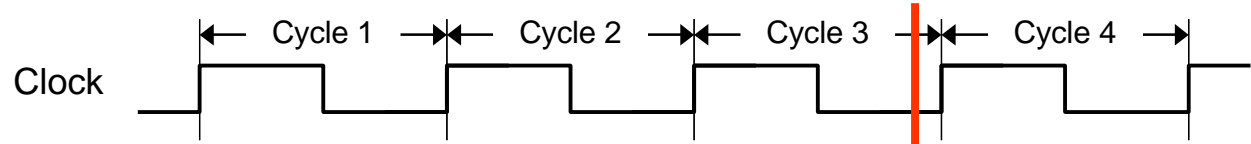
Cycle 1:	0x000: <code>irmovl \$0x100,%ebx</code> # <code>%ebx &lt;-- 0x100</code>
Cycle 2:	0x006: <code>irmovl \$0x200,%edx</code> # <code>%edx &lt;-- 0x200</code>
Cycle 3:	0x00c: <code>addl %edx,%ebx</code> # <code>%ebx &lt;-- 0x300</code> CC <-- 000
Cycle 4:	0x00e: <code>je dest</code> # Not taken



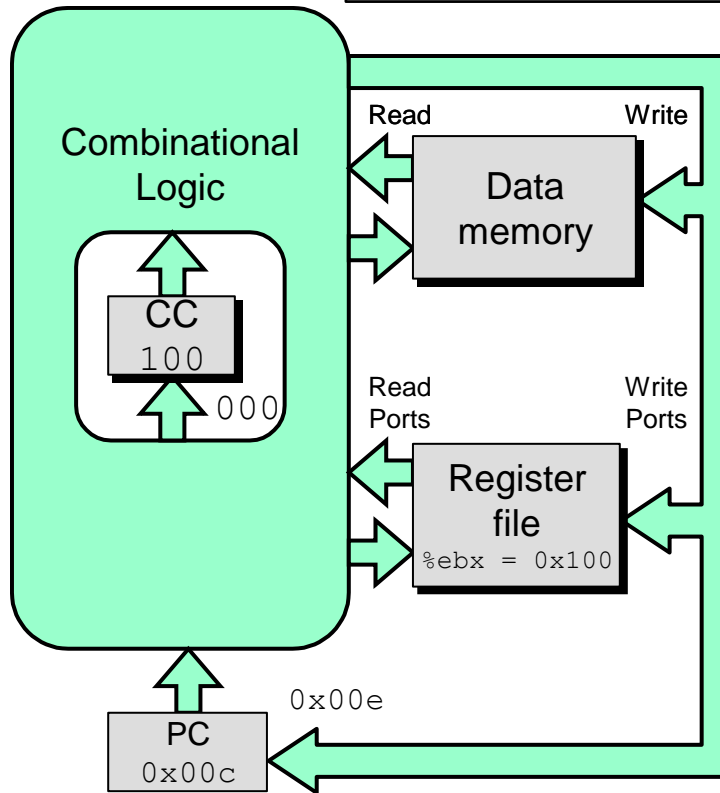
- **state set according to second `irmovl` instruction**
- **combinational logic starting to react to state changes**

SEQ

# Operation #3



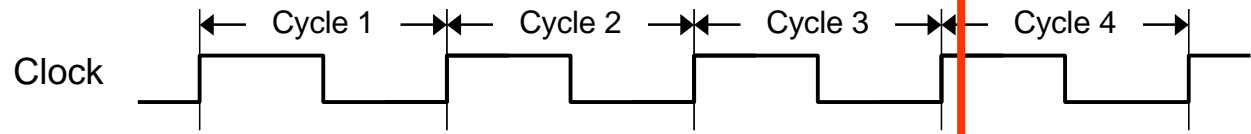
Cycle 1:	0x000: <code>irmovl \$0x100,%ebx</code> # <code>%ebx &lt;-- 0x100</code>
Cycle 2:	0x006: <code>irmovl \$0x200,%edx</code> # <code>%edx &lt;-- 0x200</code>
Cycle 3:	0x00c: <code>addl %edx,%ebx</code> # <code>%ebx &lt;-- 0x300</code> CC <-- 000
Cycle 4:	0x00e: <code>je dest</code> # Not taken



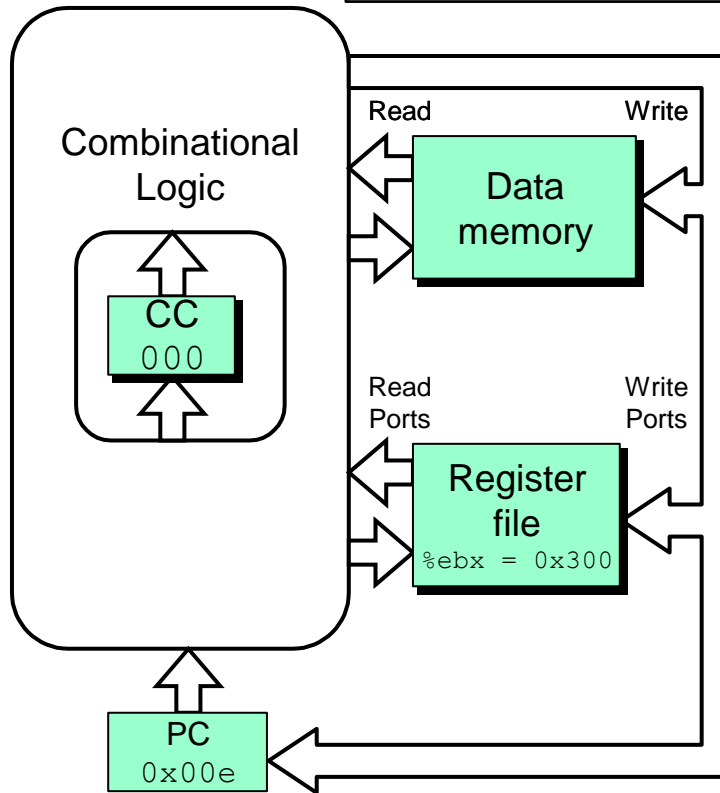
- state set according to second `irmovl` instruction
- combinational logic generates results for `addl` instruction



SEQ

Operation  
#4

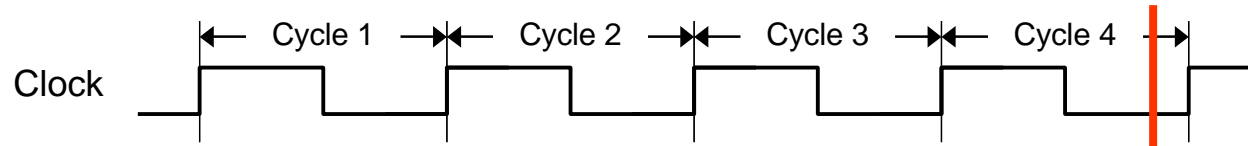
Cycle 1:	0x000: <code>irmovl \$0x100,%ebx</code> # <code>%ebx &lt;-- 0x100</code>
Cycle 2:	0x006: <code>irmovl \$0x200,%edx</code> # <code>%edx &lt;-- 0x200</code>
Cycle 3:	0x00c: <code>addl %edx,%ebx</code> # <code>%ebx &lt;-- 0x300</code> CC <-- 000
Cycle 4:	0x00e: <code>je dest</code> # Not taken



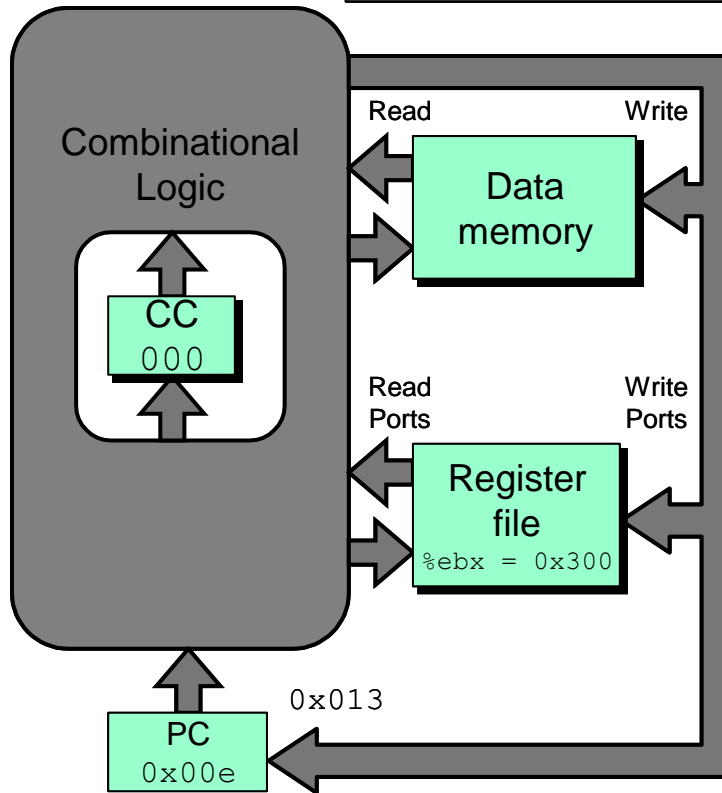
- **state set according to `addl` instruction**
- **combinational logic starting to react to state changes**

# SEQ

## Operation #5



Cycle 1:	0x000: <code>irmovl \$0x100,%ebx</code> # <code>%ebx &lt;-- 0x100</code>
Cycle 2:	0x006: <code>irmovl \$0x200,%edx</code> # <code>%edx &lt;-- 0x200</code>
Cycle 3:	0x00c: <code>addl %edx,%ebx</code> # <code>%ebx &lt;-- 0x300</code> CC <-- 000
Cycle 4:	0x00e: <code>je dest</code> # Not taken



- 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