

# Processor Architecture: Sequential

COMP402127: Introduction to Computer Systems

<https://xjtu-ics.github.io/>

Danfeng Shan

Xi'an Jiaotong University

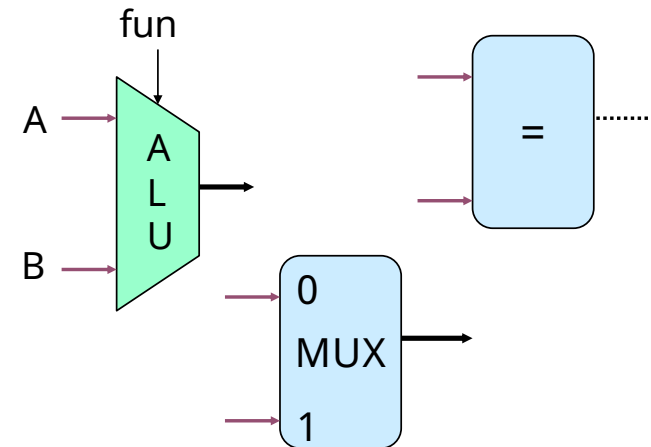
# Today

- Overview
- Y86-64 Instruction Set Architecture
- Logic Design
- **Sequential Implementation**
- Pipelined Implementation (Part A)
- Pipelined Implementation (Part B)

# Building Blocks

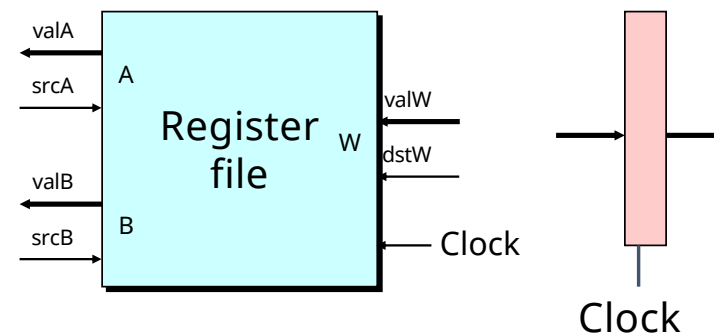
## □ Combinational Logic

- ◆ 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



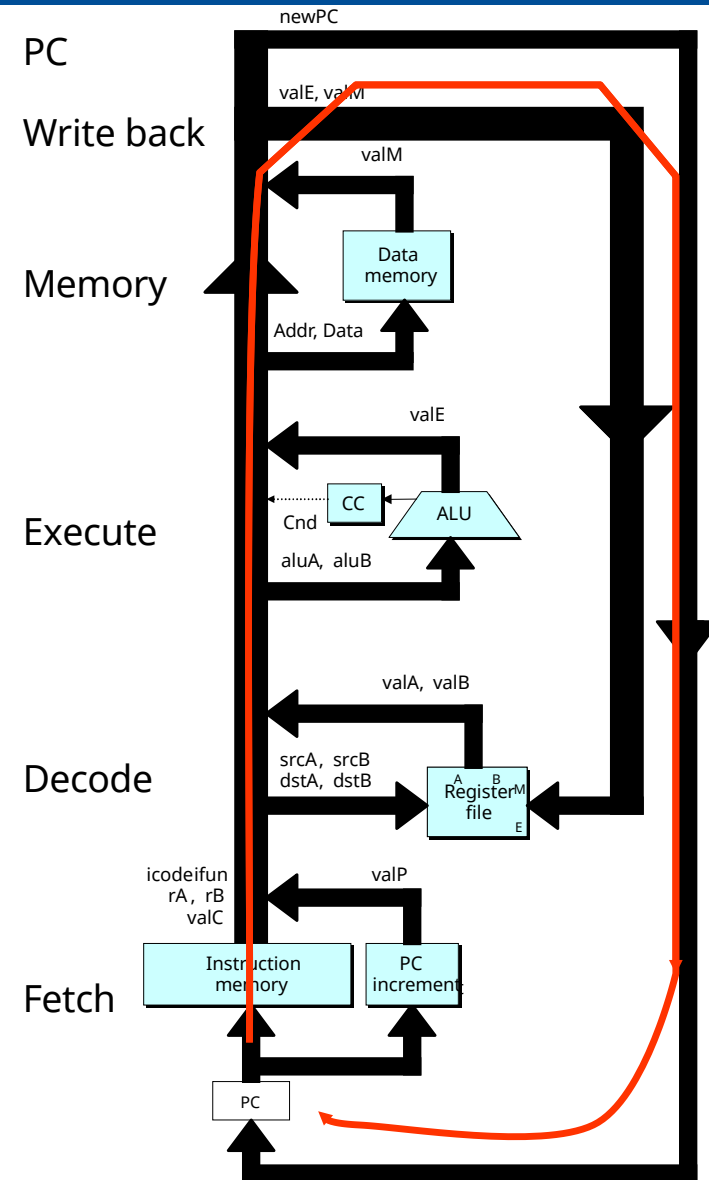
# SEQ Hardware Structure

## 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



# SEQ Hardware Structure

## Fetch

- ◆ Read instruction from instruction memory

## Decode

- ◆ Read program registers

## Execute

- ◆ Compute value or address

## Memory

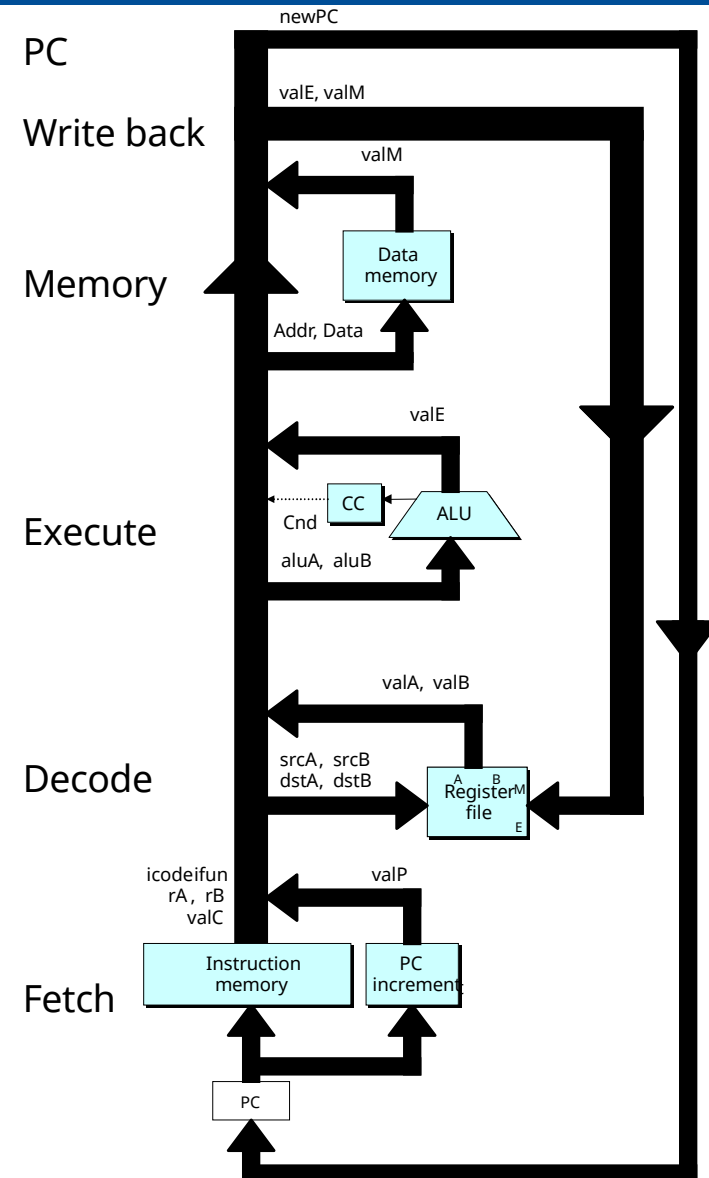
- ◆ Read or write data

## Write Back

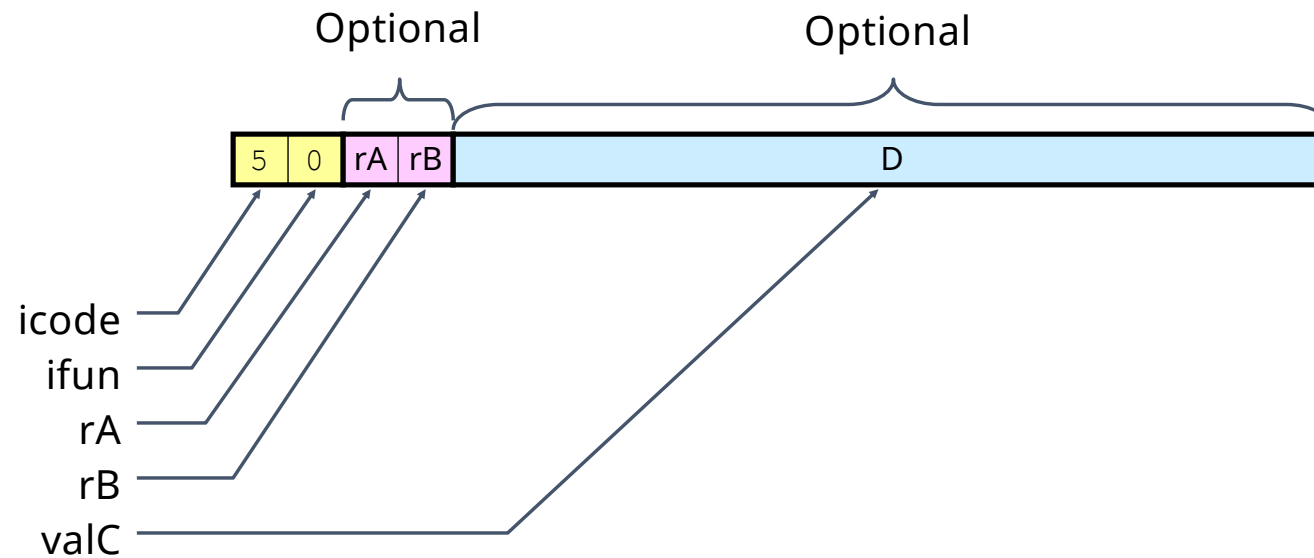
- ◆ Write program registers

## 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

OPq 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

	$OPq\ rA, rB$	
Fetch	$icode:ifun \leftarrow M_1[PC]$	Read instruction byte
	$rA:rB \leftarrow M_1[PC+1]$	Read register byte
	$valP \leftarrow PC+2$	Compute next PC
Decode	$valA \leftarrow R[rA]$	Read operand A
	$valB \leftarrow R[rB]$	Read operand B
Execute	$valE \leftarrow valB\ OP\ valA$ Set CC	Perform ALU operation Set condition code register
Memory		
Write back	$R[rB] \leftarrow valE$	Write back result
PC update	$PC \leftarrow valP$	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	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$	Read instruction byte Read register byte Read displacement D Compute next PC
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$	Read operand A Read operand B
Execute	$valE \leftarrow valB + valC$	Compute effective address
Memory	$M_8[valE] \leftarrow valA$	Write value to memory
Write back		
PC update	$PC \leftarrow valP$	Update PC

- ◆ Use ALU for address computation

# Executing popq

popq rA

b	0	rA	8
---	---	----	---

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

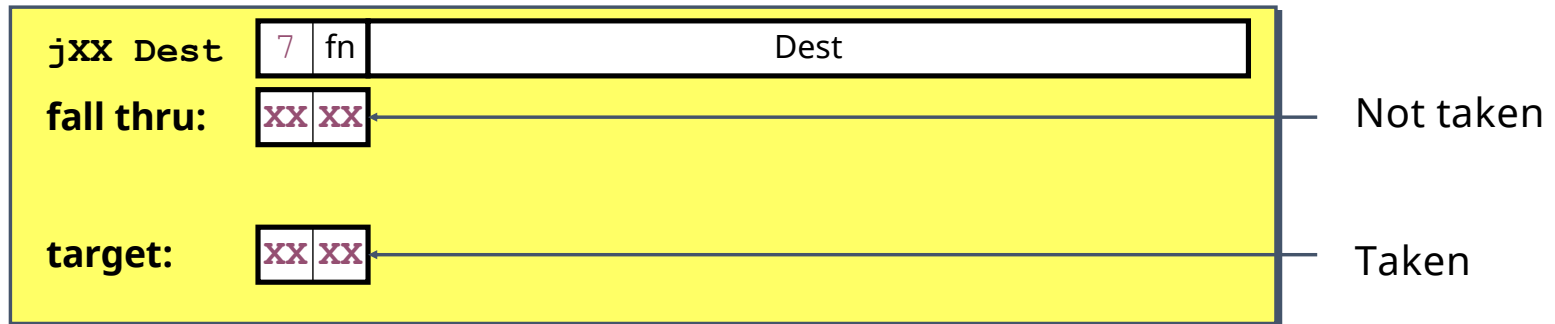
- ◆ Increment PC by 2

## Stage Computation: Cond. Move

	<b>cmovXX rA, rB</b>	
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) 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

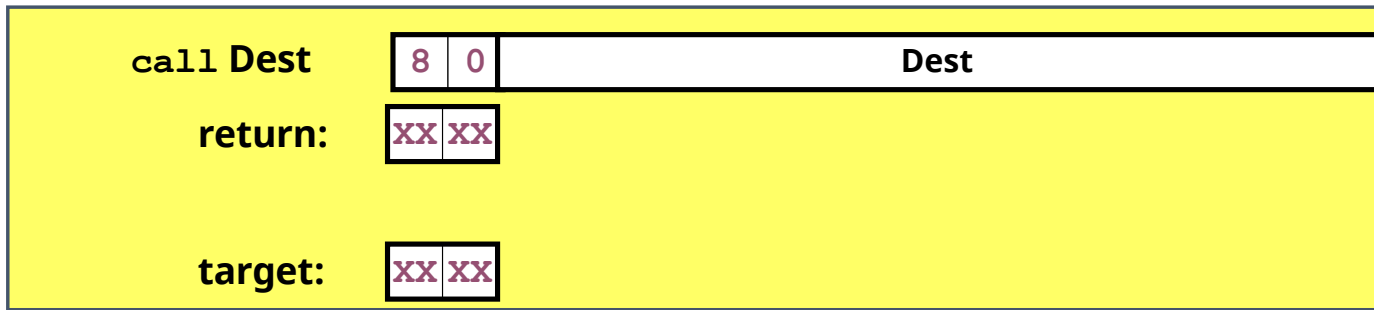
- ◆ Set PC to `Dest` if branch taken or to incremented PC if not branch

# Stage Computation: Jumps

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

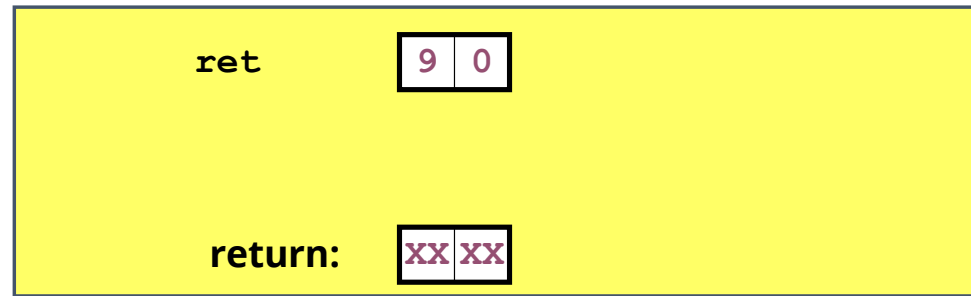
- ◆ Set PC to Dest

# Stage Computation: call

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

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

# Computation Steps

		OP <sub>q</sub> rA, rB	
Fetch	icode,ifun	$icode:ifun \leftarrow M_1[PC]$	Read instruction byte
	rA,rB	$rA:rB \leftarrow M_1[PC+1]$	Read register byte
	valC		[Read constant word]
	valP	$valP \leftarrow PC+2$	Compute next PC
Decode	valA, srcA	$valA \leftarrow R[rA]$	Read operand A
	valB, srcB	$valB \leftarrow R[rB]$	Read operand B
Execute	valE	$valE \leftarrow valB \text{ OP } valA$	Perform ALU operation
	Cond code	Set CC	Set/use cond. code reg
Memory	valM		[Memory read/write]
Write back	dstE	$R[rB] \leftarrow valE$	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	$PC \leftarrow valP$	Update PC

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

# Computation Steps

		call Dest	
Fetch	icode,ifun	$icode:ifun \leftarrow M_1[PC]$	Read instruction byte
	rA,rB		[Read register byte]
	valC	$valC \leftarrow M_8[PC+1]$	Read constant word
	valP	$valP \leftarrow PC+9$	Compute next PC
Decode	valA, srcA		[Read operand A]
	valB, srcB	$valB \leftarrow R[\%rsp]$	Read operand B
Execute	valE	$valE \leftarrow valB + -8$	Perform ALU operation
	Cond code		[Set /use cond. code reg]
Memory	valM	$M_8[valE] \leftarrow valP$	Memory read/write
Write back	dstE	$R[\%rsp] \leftarrow valE$	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	$PC \leftarrow valC$	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
◆ Cnd	Branch/move flag

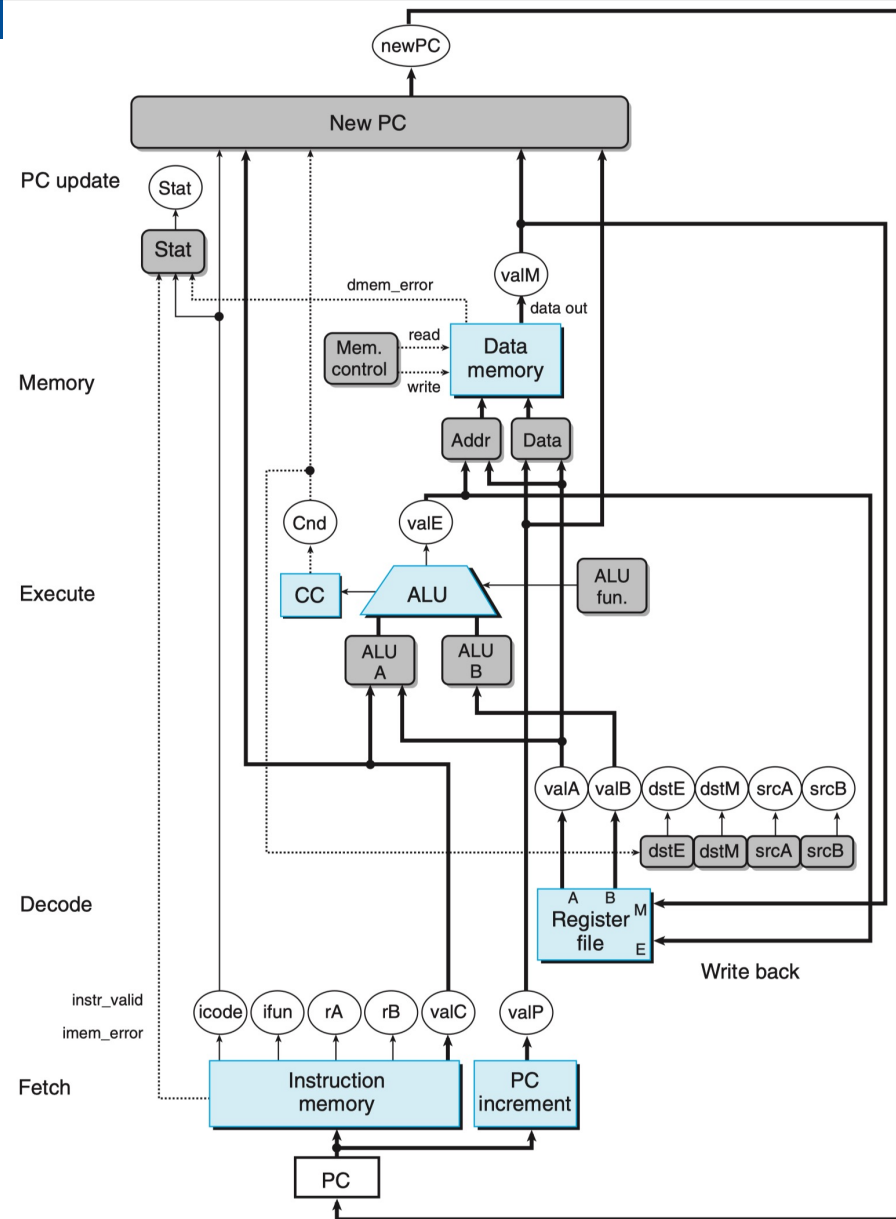
## □ Memory

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

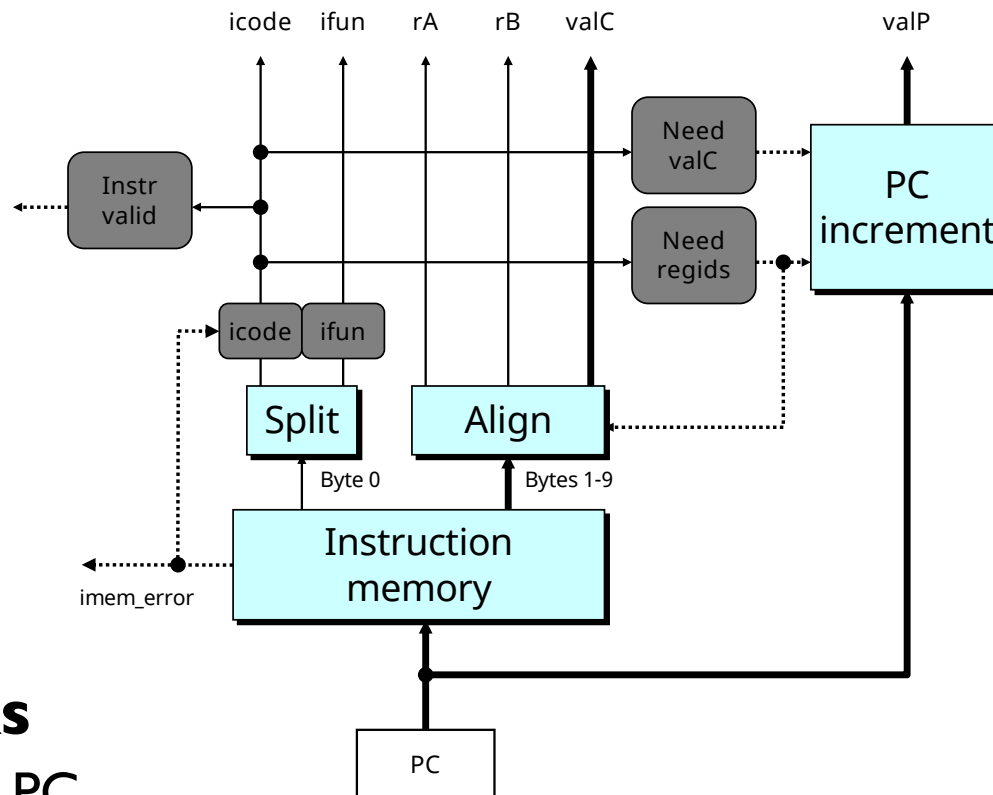
# SEQ Hardware

## Key

- ◆ Blue boxes: predesigned hardware blocks
  - E.g., memories, ALU
- ◆ Gray boxes: control logic
  - Selection or compute Boolean function
  - Describe in HCL
- ◆ White ovals: labels for signals
  - Wire names
- ◆ Thick lines: 64-bit word values
- ◆ Thin lines: 4-8 bit values
- ◆ Dotted lines: 1-bit values



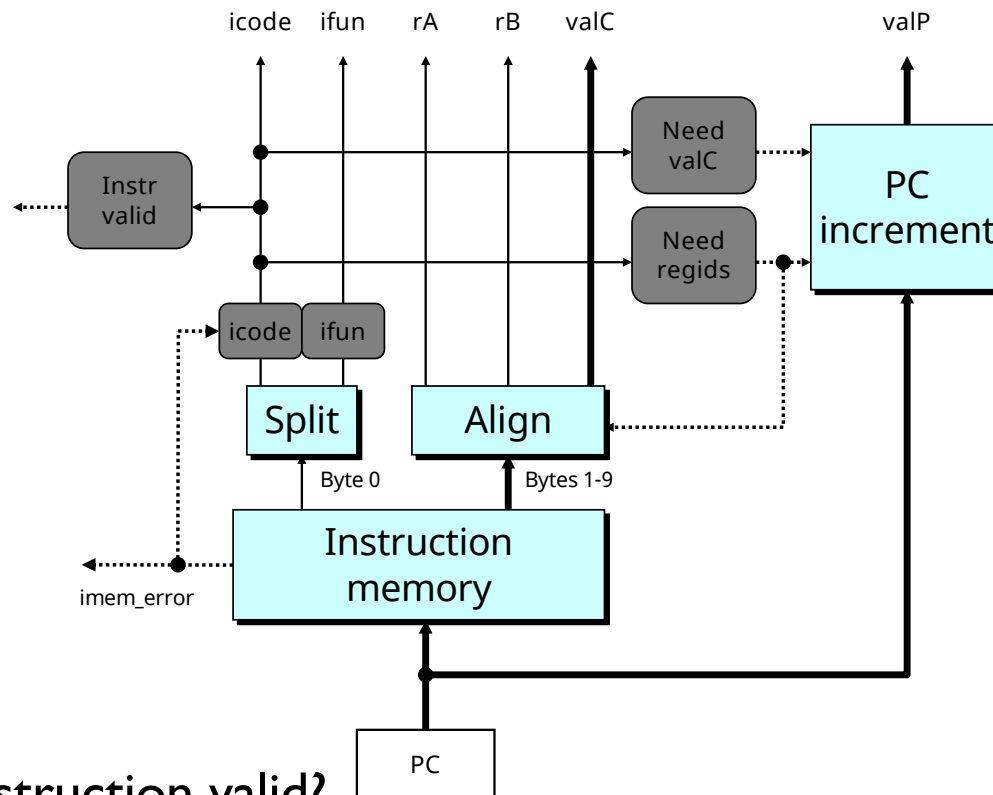
# Fetch Logic



## 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



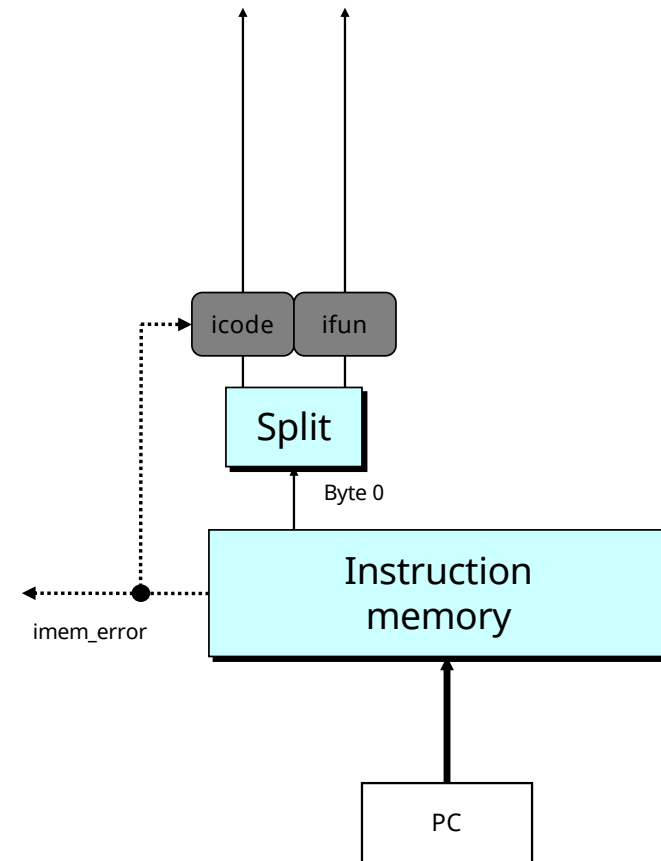
## Control Logic

- ◆ `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

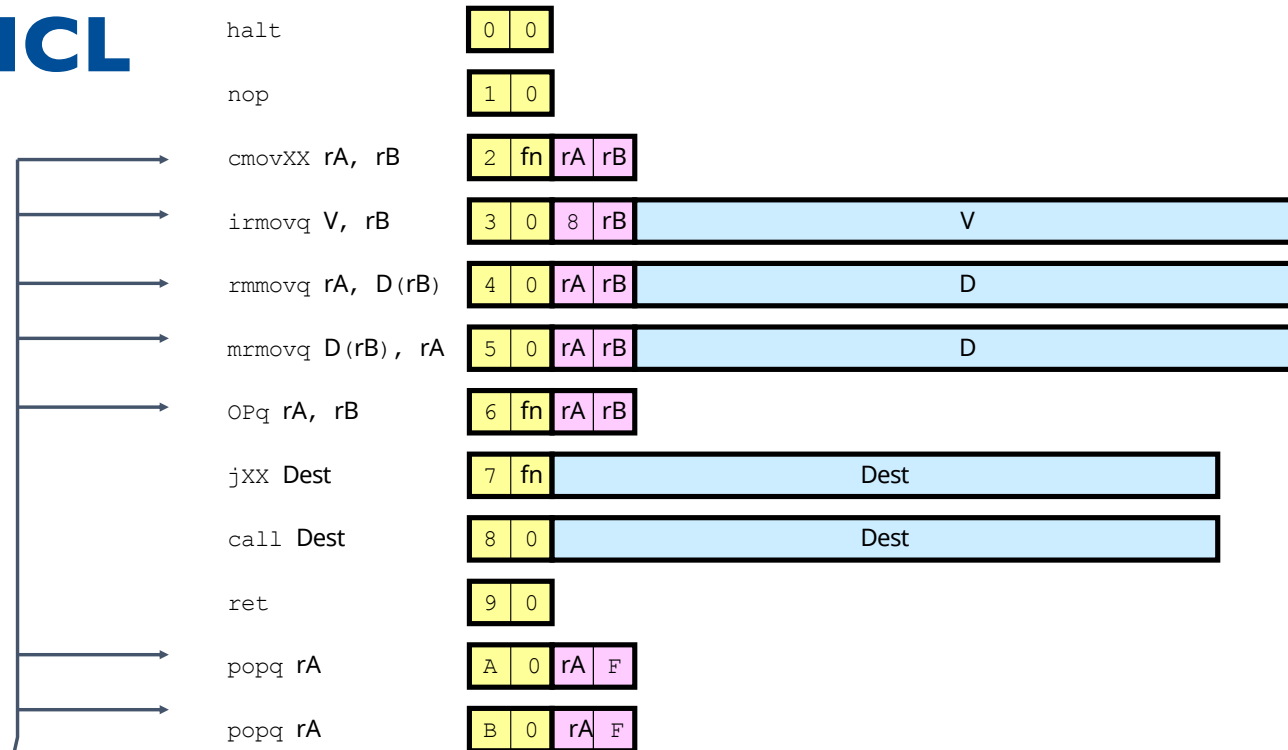
```
# Determine instruction code
int icode = [
    imem_error: INOP;
    1: imem_icode;
];

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



Name	Value (hex)	Meaning
IHALT	0	Code for halt instruction
INOP	1	Code for nop instruction
IRRMVQ	2	Code for rrmovq instruction
IIRMOVQ	3	Code for irmovq instruction
IRMMOVQ	4	Code for rmmovq instruction
IMRMVQ	5	Code for mrmovq instruction
IOPL	6	Code for integer operation instructions
IJXX	7	Code for jump instructions
ICALL	8	Code for call instruction
IRET	9	Code for ret instruction
IPUSHQ	A	Code for pushq instruction
IPOPQ	B	Code for popq instruction
FNONE	0	Default function code
RESP	4	Register ID for %rsp
RNONE	F	Indicates no register file access
ALUADD	0	Function for addition operation
SAOK	1	Status code for normal operation
SADR	2	Status code for address exception
SINS	3	Status code for illegal instruction exception
SHLT	4	Status code for halt

# Fetch Control Logic in HCL



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

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

# Decode and Write-back Logic

## □ Register File

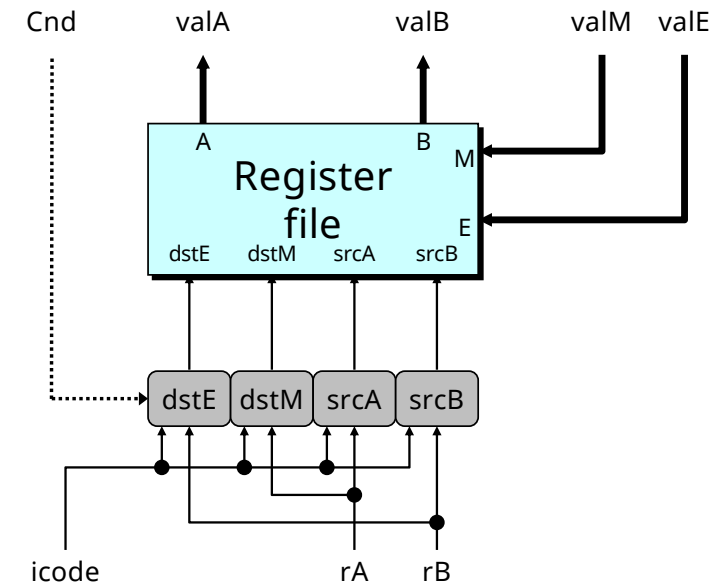
- ◆ Read ports A, B
- ◆ Write ports E, M
- ◆ Addresses are register IDs or 15 (0xF) (no access)

## □ 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

	<code>OPq rA, rB</code>	
Decode	<code>valA ← R[rA]</code>	Read operand A
	<code>cmovXX rA, rB</code>	
Decode	<code>valA ← R[rA]</code>	Read operand A
	<code>rmmovq rA, D(rB)</code>	
Decode	<code>valA ← R[rA]</code>	Read operand A
	<code>popq rA</code>	
Decode	<code>valA ← R[%rsp]</code>	Read stack pointer
	<code>jXX Dest</code>	
Decode		No operand
	<code>call Dest</code>	
Decode		No operand
	<code>ret</code>	
Decode	<code>valA ← R[%rsp]</code>	Read stack pointer

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

## E Destination

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

```

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

```

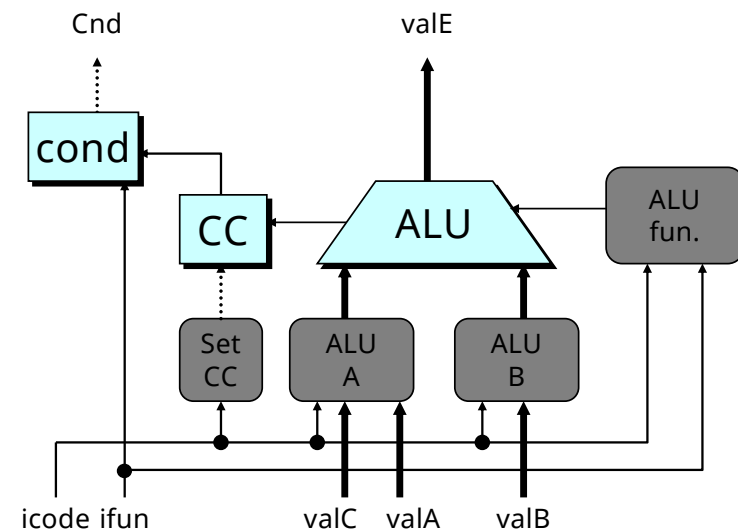
# Execute Logic

## □ 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

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

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

# ALU Operation

	<code>OP1 rA, rB</code>	
Execute	$valE \leftarrow valB \text{ OP } valA$	Perform ALU operation
	<code>cmovXX rA, rB</code>	
Execute	$valE \leftarrow 0 + valA$	Pass valA through ALU
	<code>rmmovl rA, D(rB)</code>	
Execute	$valE \leftarrow valB + valC$	Compute effective address
	<code>popq rA</code>	
Execute	$valE \leftarrow valB + 8$	Increment stack pointer
	<code>jXX Dest</code>	
Execute		No operation
	<code>call Dest</code>	
Execute	$valE \leftarrow valB + -8$	Decrement stack pointer
	<code>ret</code>	
Execute	$valE \leftarrow valB + 8$	Increment stack pointer

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

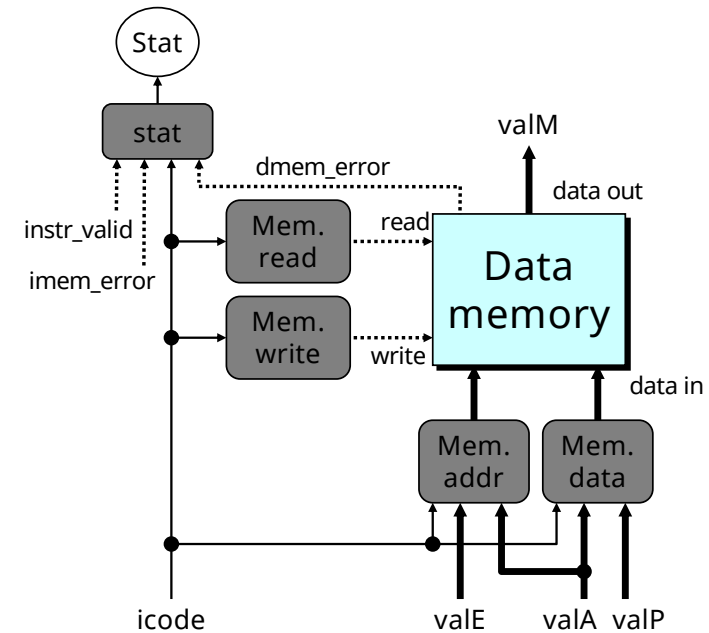
# Memory Logic

## □ Memory

- ◆ Reads or writes memory word

## □ Control Logic

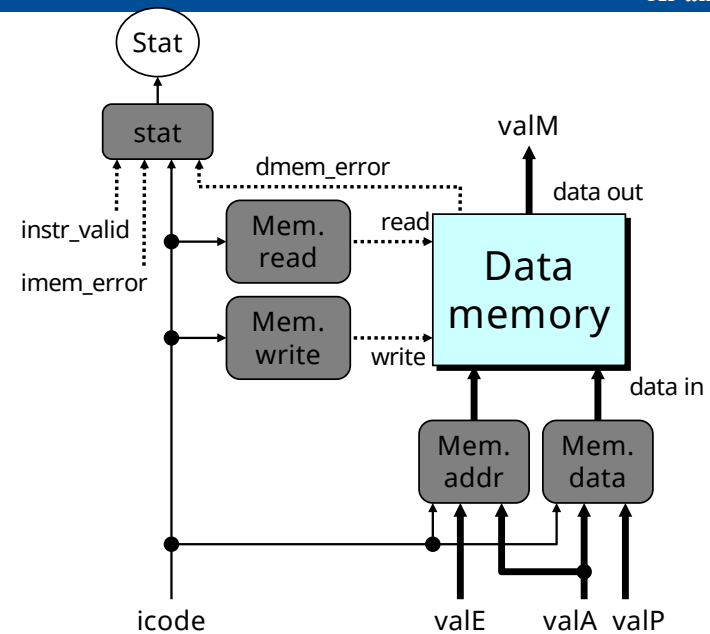
- ◆ stat: What is instruction status?
- ◆ Mem. read: should word be read?
- ◆ Mem. write: should word be written?
- ◆ Mem. addr.: Select address
- ◆ Mem. data.: Select data



# 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;
];
```

Name	Value (hex)	Meaning
IHALT	0	Code for halt instruction
INOP	1	Code for nop instruction
IRRMVQ	2	Code for rrmovq instruction
IIRMOVQ	3	Code for irmovq instruction
IRMMOVQ	4	Code for rmmovq instruction
IMRMVQ	5	Code for mrmovq instruction
IOPL	6	Code for integer operation instructions
IJXX	7	Code for jump instructions
ICALL	8	Code for call instruction
IRET	9	Code for ret instruction
IPUSHQ	A	Code for pushq instruction
IPOPQ	B	Code for popq instruction
FNONE	0	Default function code
RESP	4	Register ID for %rsp
RNONE	F	Indicates no register file access
ALUADD	0	Function for addition operation
SAOK	1	Status code for normal operation
SADR	2	Status code for address exception
SINS	3	Status code for illegal instruction exception
SHLT	4	Status code for halt

# Memory Address

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

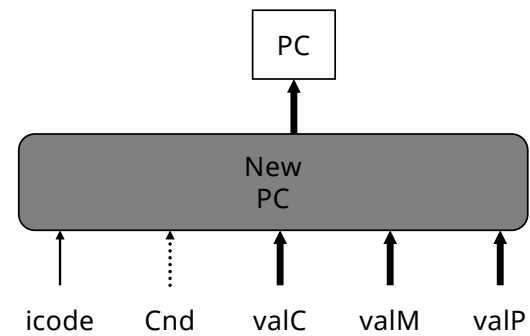
Memory	<code>OPq rA, rB</code>	No operation
Memory	<code>rmmovq rA, D(rB)</code> $M_8[valE] \leftarrow valA$	Write value to memory
Memory	<code>popq rA</code> $valM \leftarrow M_8[valA]$	Read from stack
Memory	<code>jXX Dest</code>	No operation
Memory	<code>call Dest</code> $M_8[valE] \leftarrow valP$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

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

# PC Update Logic

## □ New PC

- ◆ Select next value of PC



# PC Update

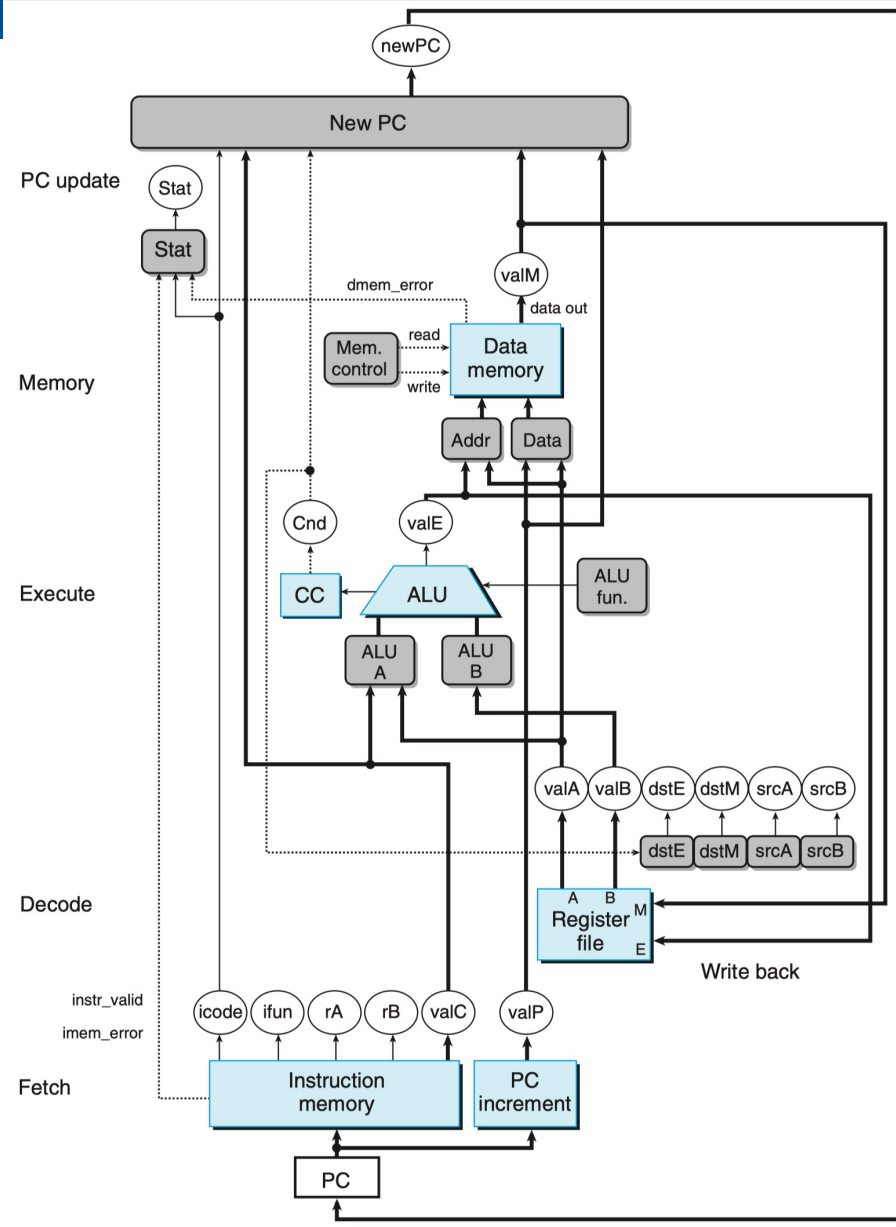
	<code>OPq rA, rB</code>	
PC update	<code>PC ← valP</code>	Update PC
	<code>rmmovq rA, D(rB)</code>	
PC update	<code>PC ← valP</code>	Update PC
	<code>popq rA</code>	
PC update	<code>PC ← valP</code>	Update PC
	<code>jXX Dest</code>	
PC update	<code>PC ← Cnd ? valC : valP</code>	Update PC
	<code>call Dest</code>	
PC update	<code>PC ← valC</code>	Set PC to destination
	<code>ret</code>	
PC update	<code>PC ← valM</code>	Set PC to return address

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

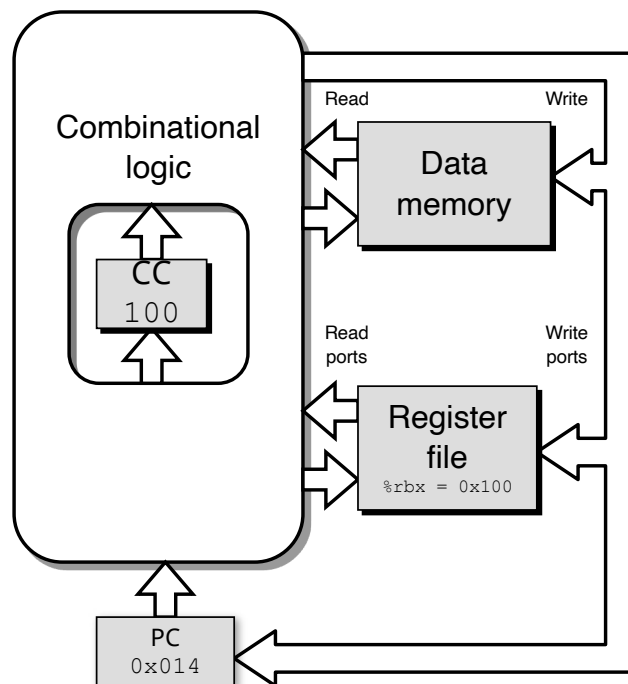
# Recall: SEQ Hardware

## Key

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



# SEQ Operation



## □ State

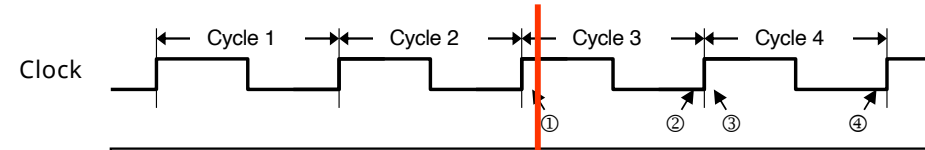
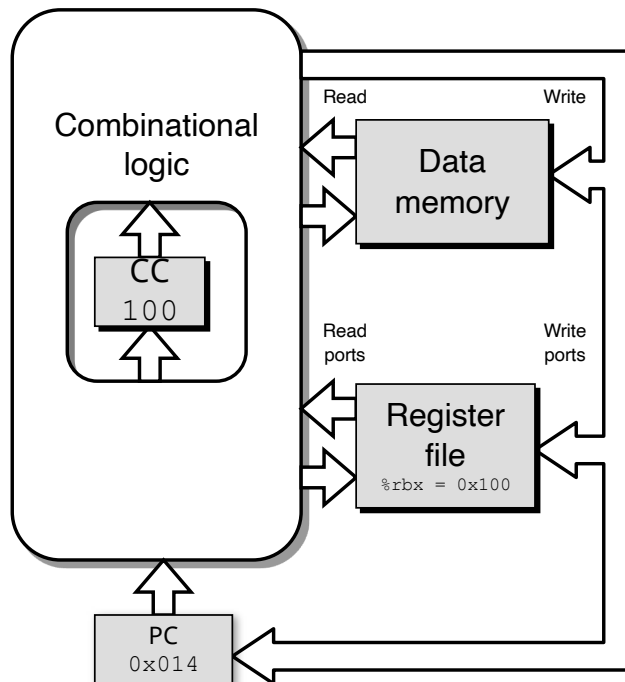
- ◆ PC register
- ◆ Cond. Code register
- ◆ Data memory
- ◆ Register file

*All updated as clock rises*

## □ Combinational Logic

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

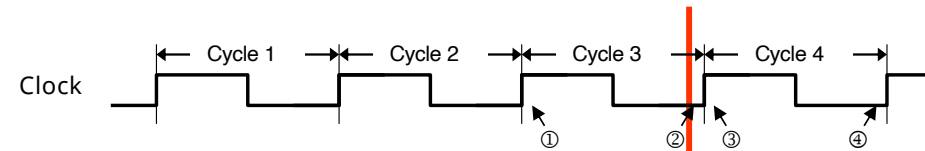
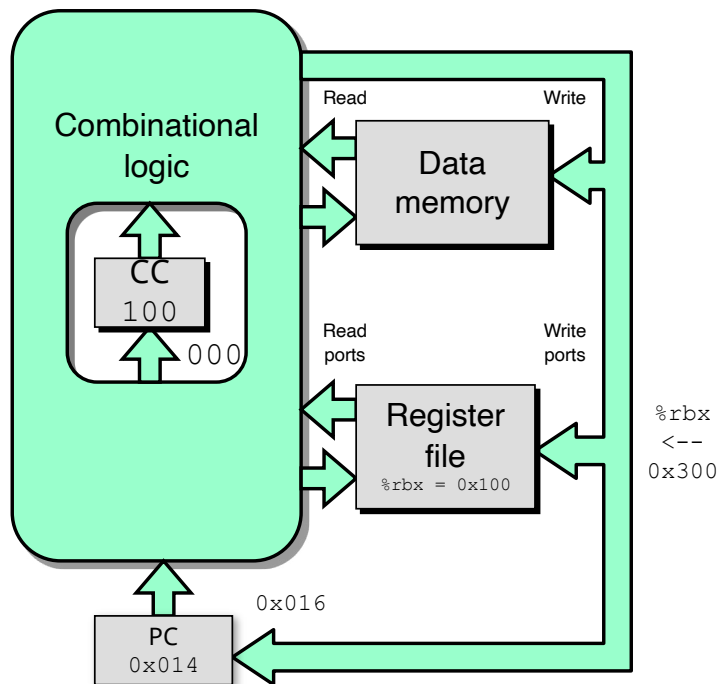
# SEQ Operation #2



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

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

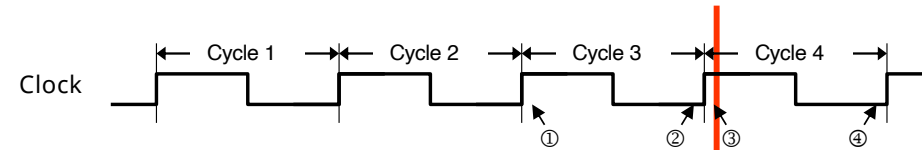
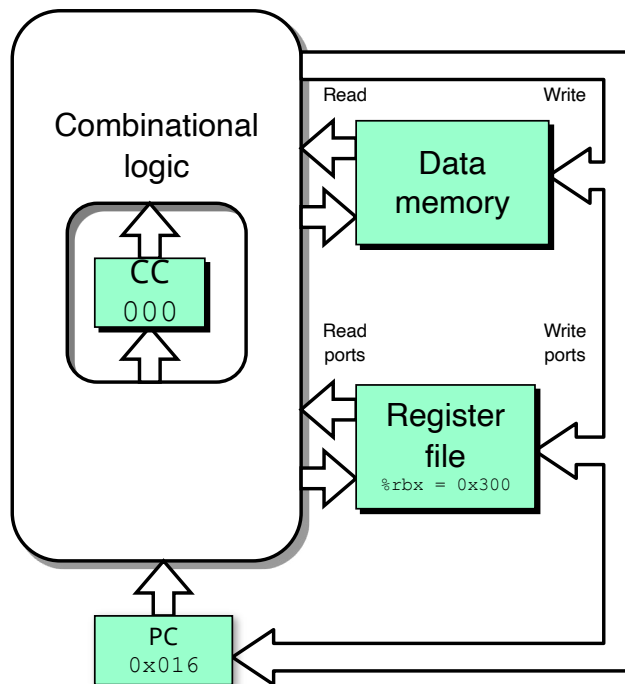
# SEQ Operation #3



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

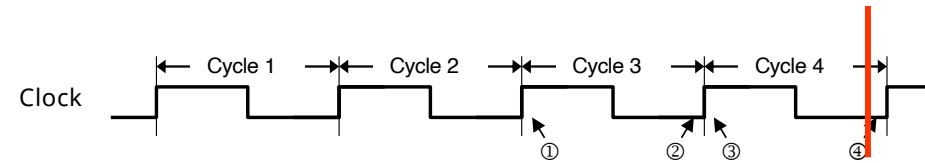
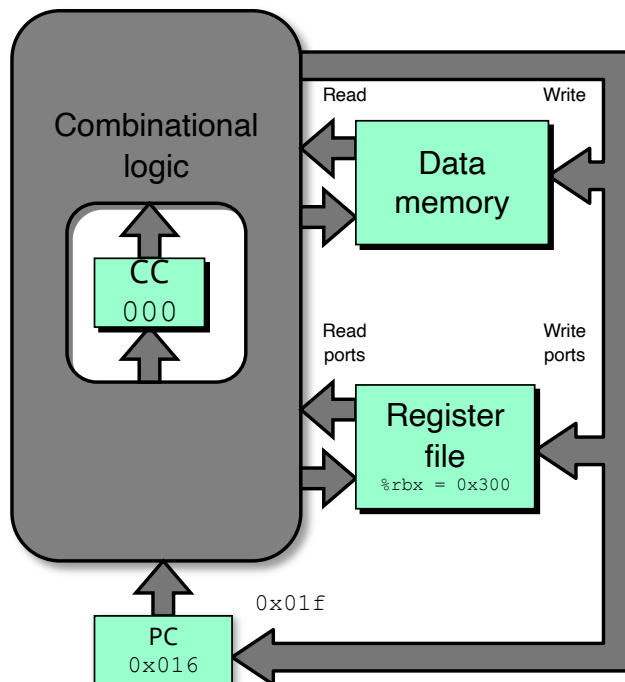
# SEQ Operation #4



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