Sequential Implementation

’20H2

송 인 식

Outline

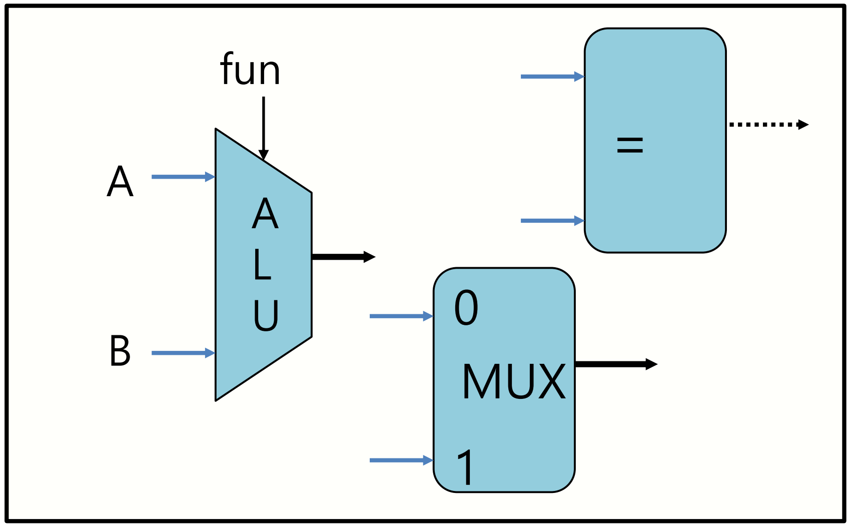
- SEQ Hardware Structure & Sequential Stages
- SEQ Stage Implementations
- SEQ Timing

Recall: Y86-64 Instruction Set

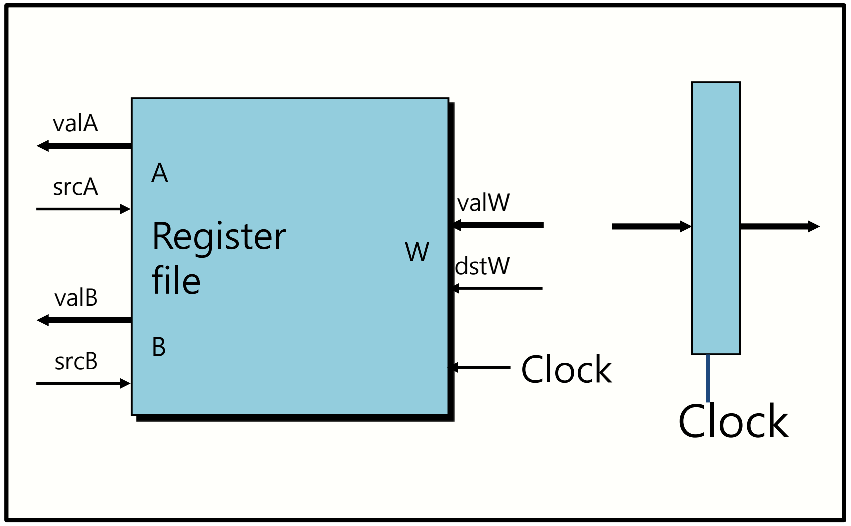
Byte	0	1	2	3	4	5	6	7	8	9
`halt`	0 0									
`nop`	1 0									
`rrmovq` rA, rB	2 0	rA rB								
`irmovq` V, rB	3 0	F rB	V							
`rmmovq` rA, D(rB)	4 0	rA rB	D							
`mrmovq` D(rB), rA	5 0	rA rB	D							
`OPq` rA, rB	6 fn	rA rB								
`jXX` Dest	7 fn	Dest								
`cmovXX` rA, rB	2 fn	rA rB						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

Building Blocks

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

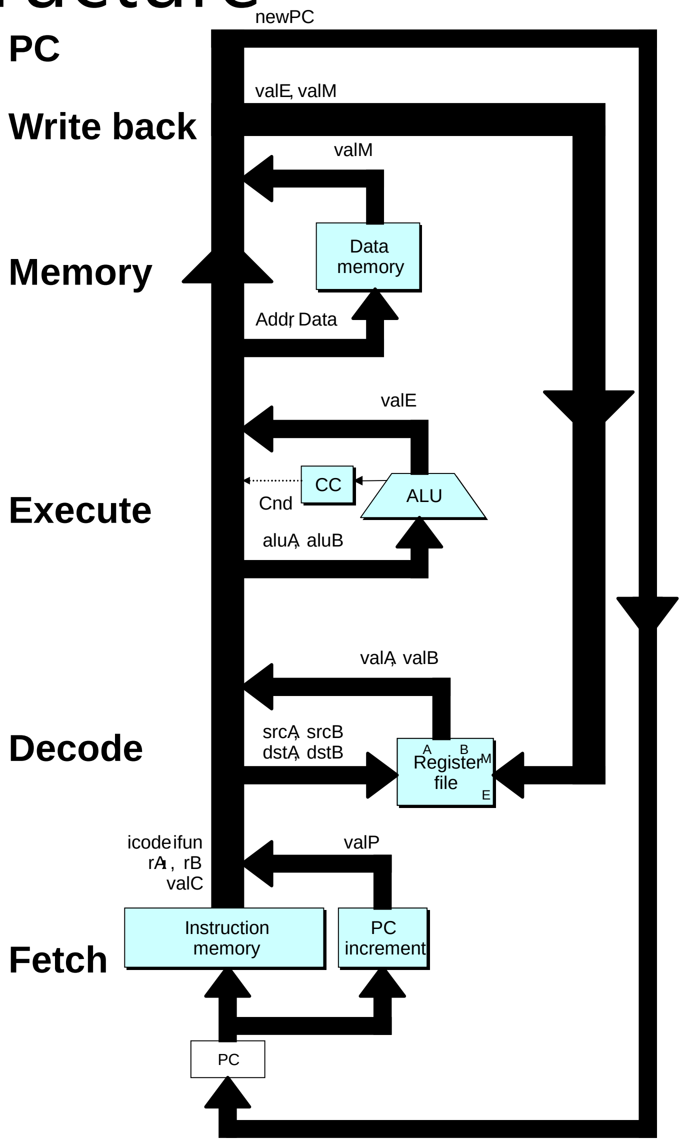


- Storage elements
  - Store bits (or states)
  - Addressable memories
  - Loaded only as clock rises



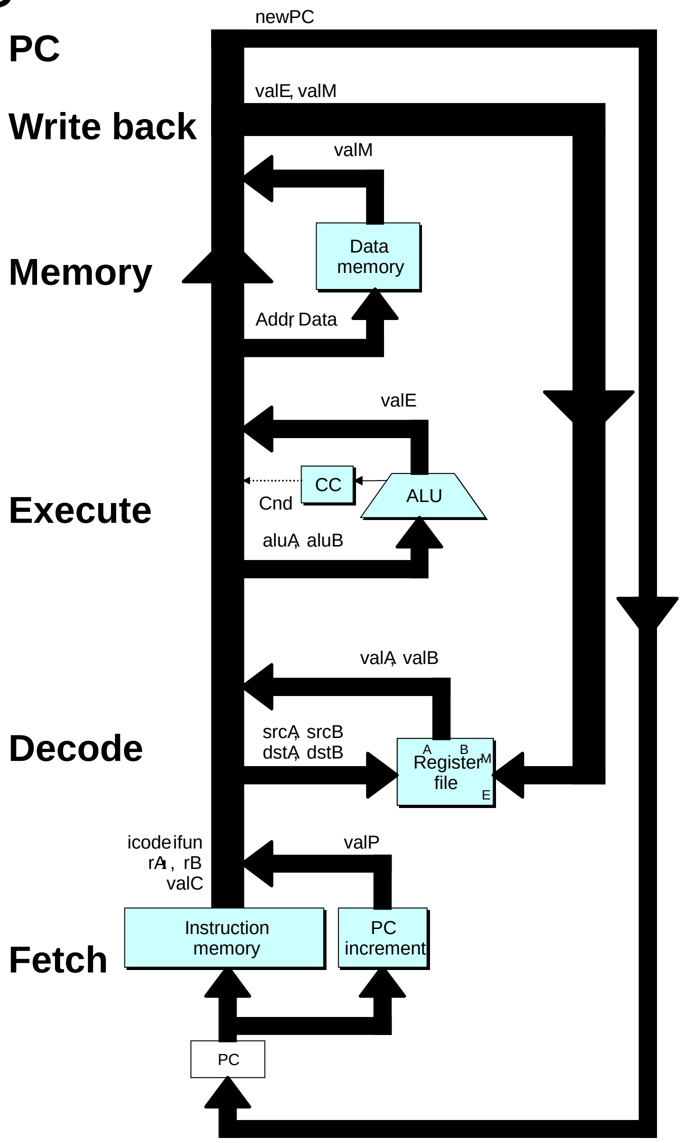
SEQ Hardware Structure

- State
  - Program counter registers (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 instructions at address specified by PC
  - Process through stages
  - Update program counter



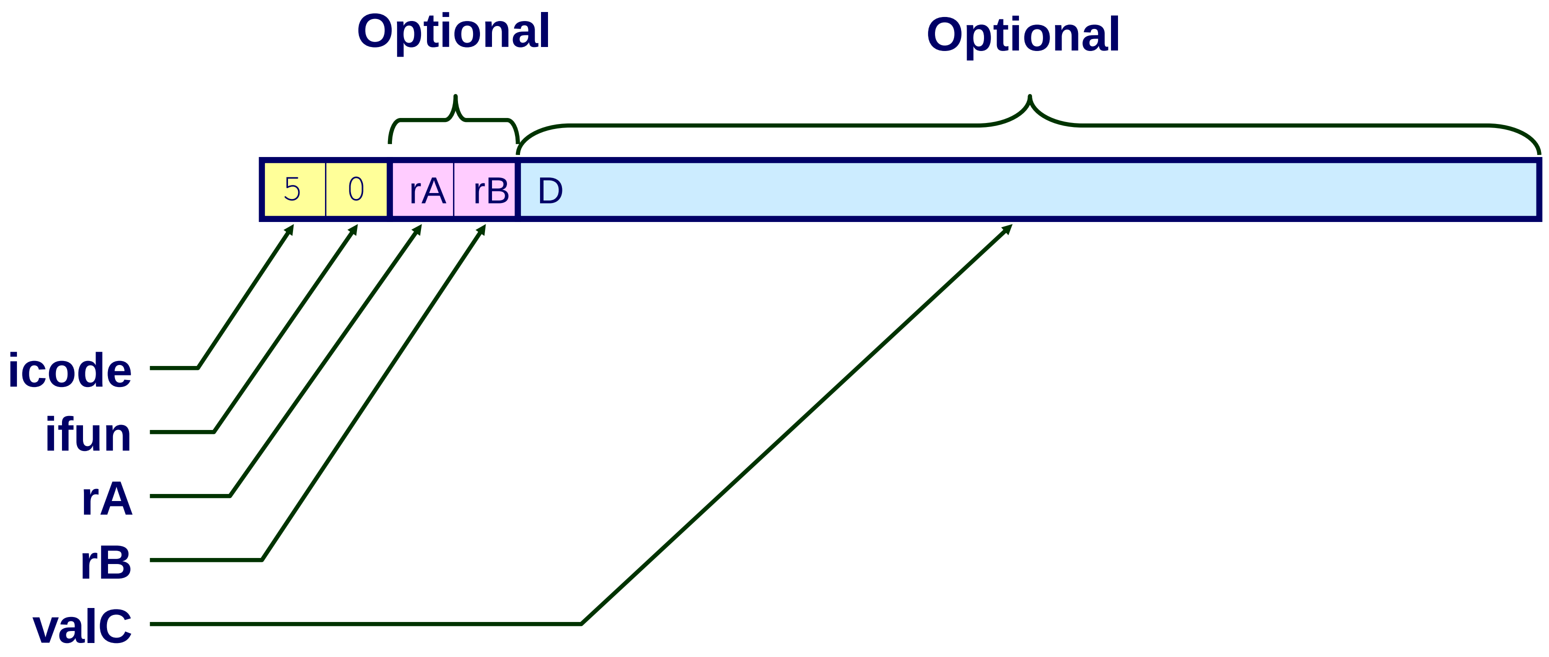
SEQ Stages

- 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: Update program counter



Instruction Decoding

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



Executing ALU Operations

```
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: ALU Operations

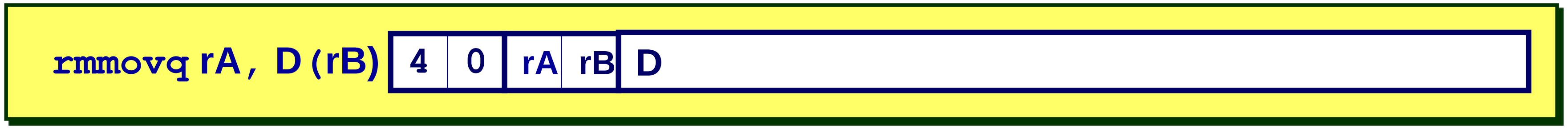
	OPq rA, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC

Decode	valA ← R[rA]
	valB ← R[rB]
		Read operand A
		Read operand B

Execute	valE ← valB OP valA Set CC	Perform ALU operation Set condition code register
Memory		
Write back	R[rB] ← valE	Write back result
PC update	PC ← valP	Update PC

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

Executing rmmovq



- 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`

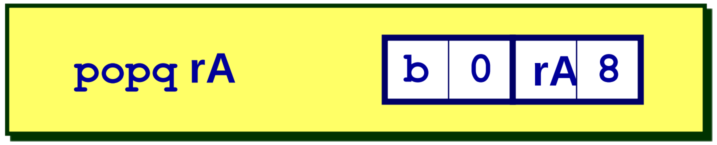
	`rmmovq` rA, D(rB)
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC+1] valC $\leftarrow$ M_8[PC+2] valP $\leftarrow$ PC+10
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[rB]

Memory	M_8[valE] $\leftarrow$ valA

PC update	PC $\leftarrow$ valP

- 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

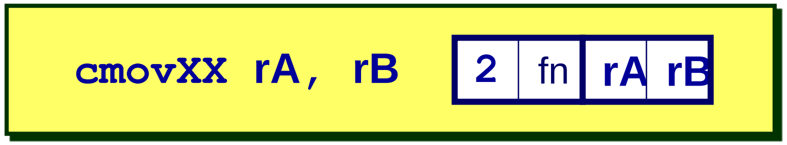
	`popq` rA	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC

Decode	valA ← R[`%rsp`]
	valB ← R[`%rsp`]
		Read stack pointer
		Read stack pointer

Execute	valE ← valB + 8	Increment stack pointer
Memory	valM ← M_8[valA]	Read from stack
Write back	R[`%rsp`] ← 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

Executing Conditional Moves



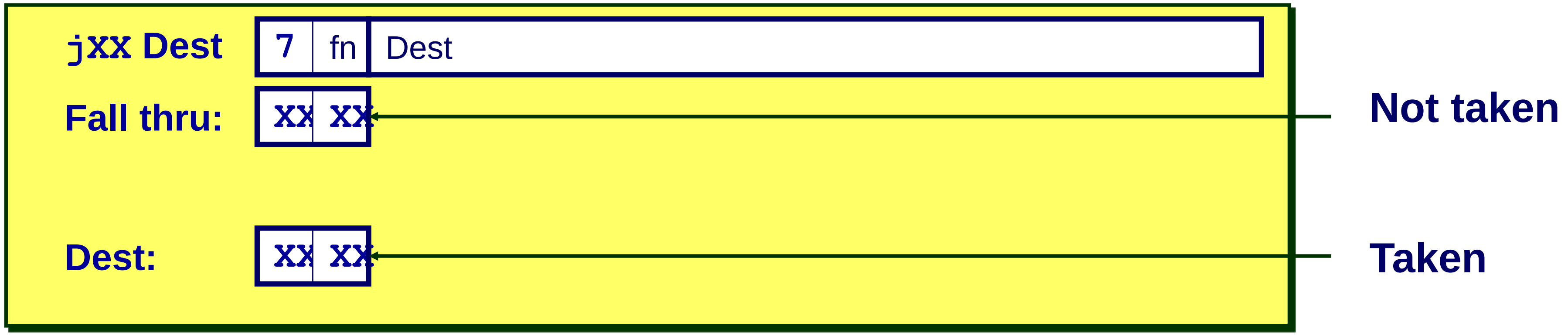
- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
- Execute
  - If !cnd, then set destination register to 0xF
- Memory
  - Do nothing
- Write back
  - Update register (or not)
- PC Update
  - Increment PC by 2

Stage Computation: Conditional Moves

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



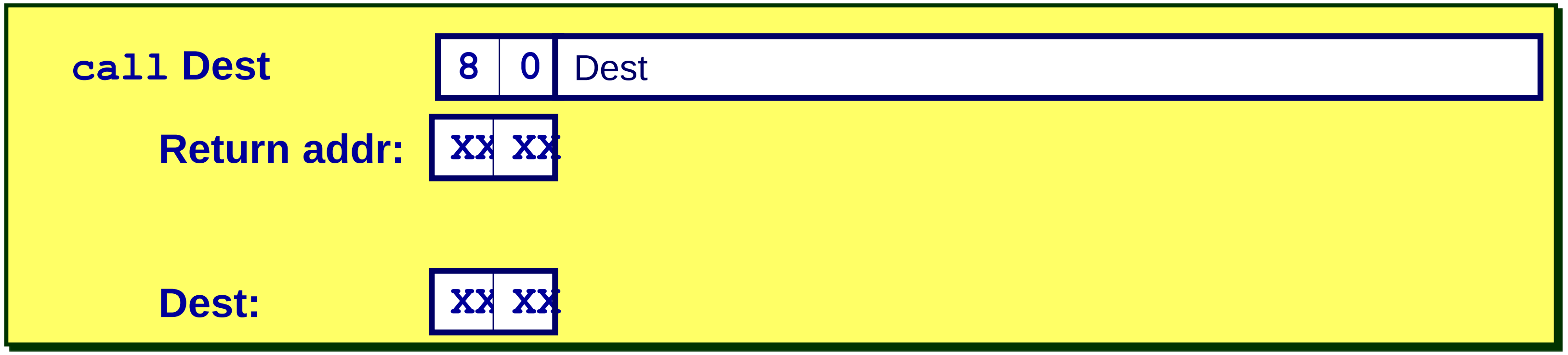
- Fetch
  - Read 9 bytes
  - Increment PC by 9
- Decode
  - Do nothing
- Execute
  - Determine whether to take branch based on jump condition and condition codes
- Memory
  - Do nothing
- Write back
  - Do nothing
- PC Update
  - Set PC to Dest if branch taken or to incremented PC if not branch

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`



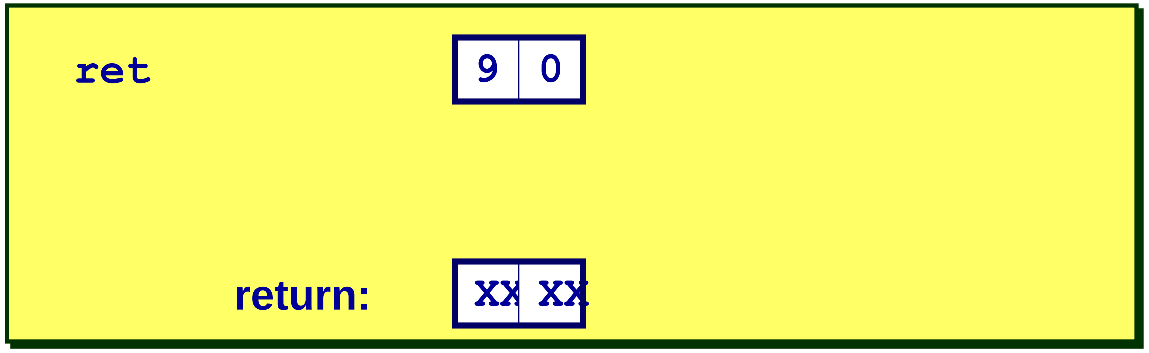
- Fetch
  - Read 9 bytes
  - Increment PC by 9
- Decode
  - Read stack pointer
- Execute
  - Decrement stack pointer by 8
- Memory
  - Write incremented PC to new value of stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to Dest

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: `call`

	`ret`	
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

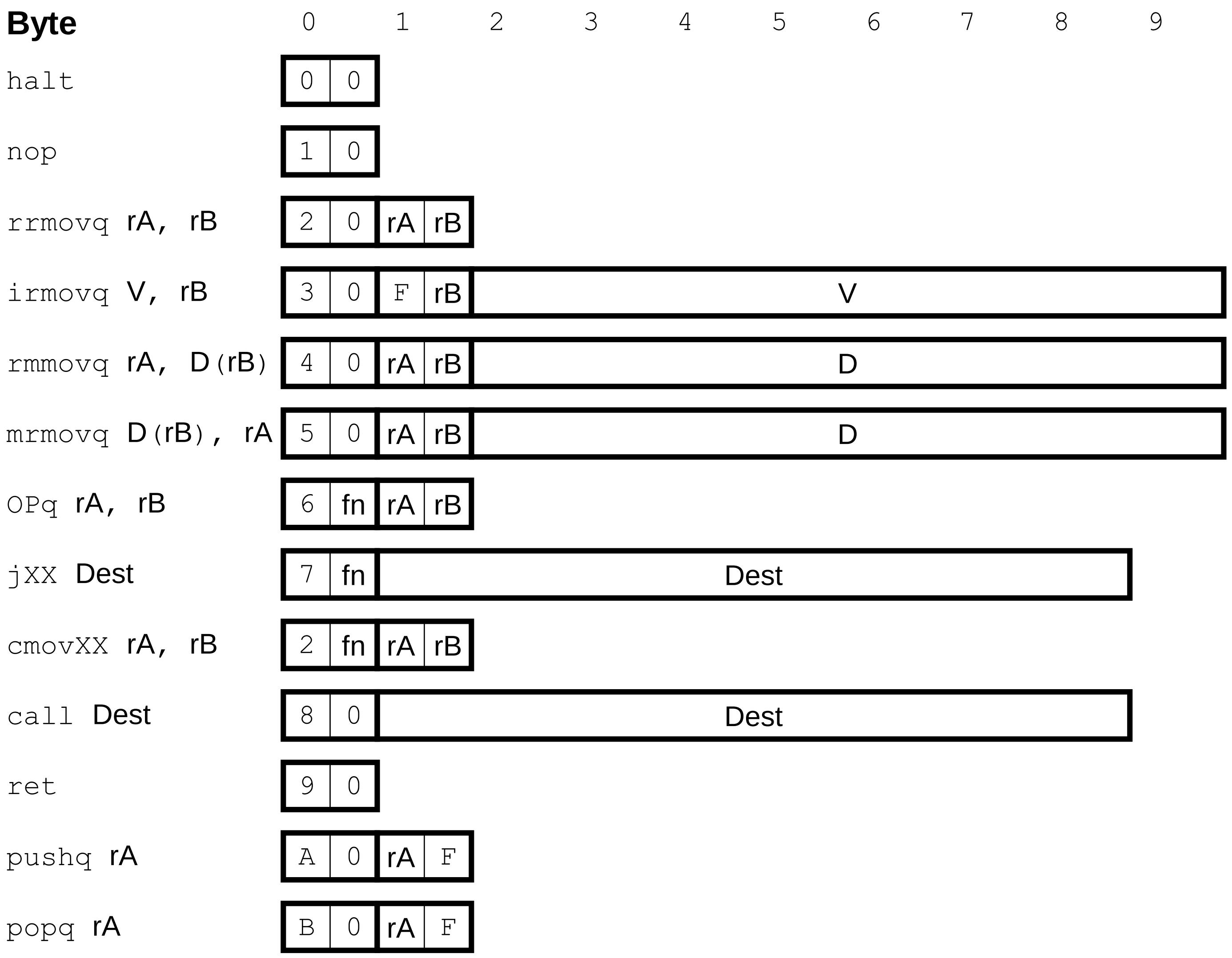
- Use ALU to increment stack pointer
- Read return address from memory

Computation Steps

		OPq 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/use cond. code reg
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

Recall: Y86-64 Instruction Set



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

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

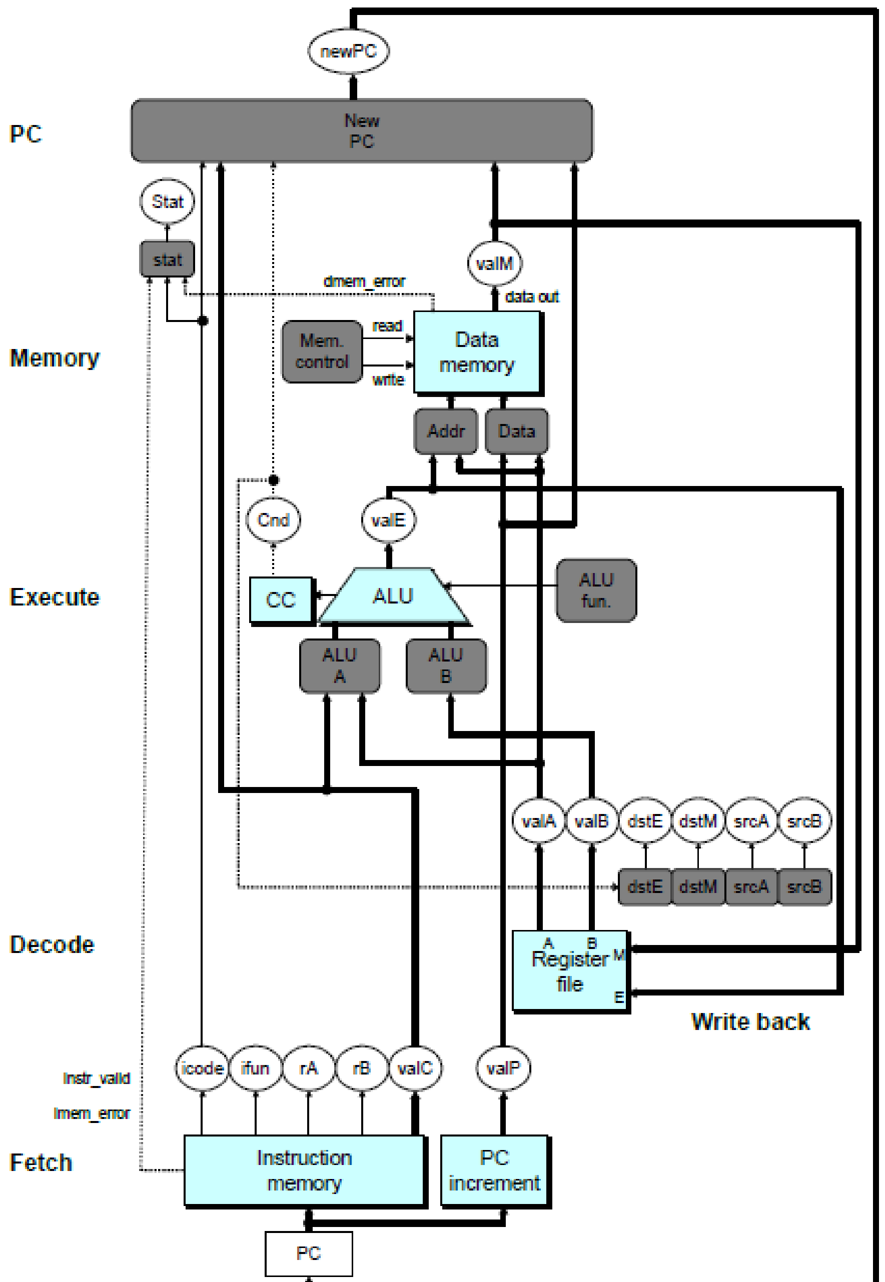
valE	ALU result
Cnd	Branch/move flag

Memory

valM	Value from memory

SEQ Hardware

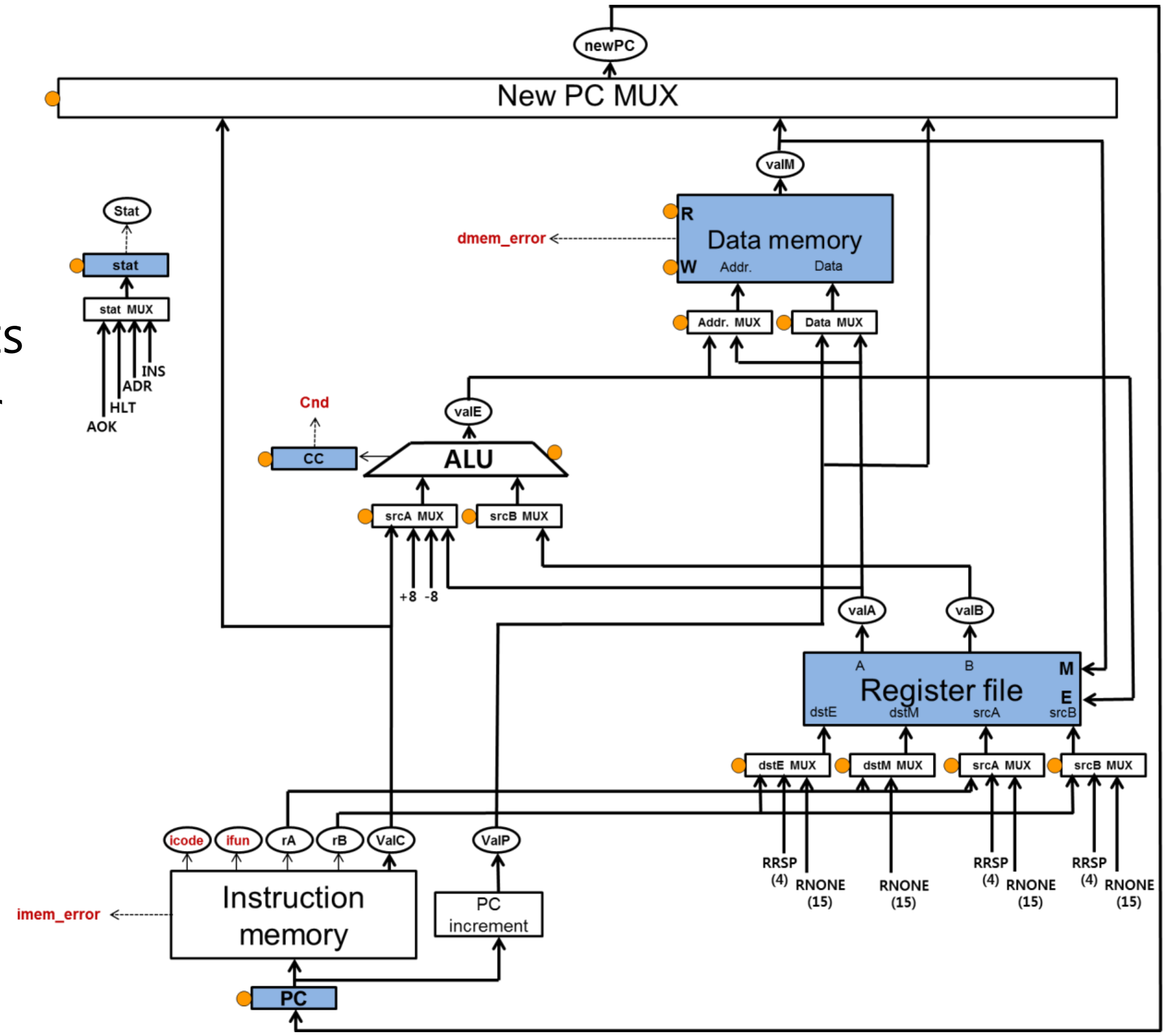
- Key
  - Blue boxes: predesigned hardware blocks
    - E.g., memories, ALU
  - Gray boxes: control logic
    - Described in HCL
  - White ovals: labels for signals
  - Thick lines: 64-bit word values
  - Thin lines: 4-8 bit values
  - Dotted lines: 1-bit values



(More Detailed) SEQ Hardware

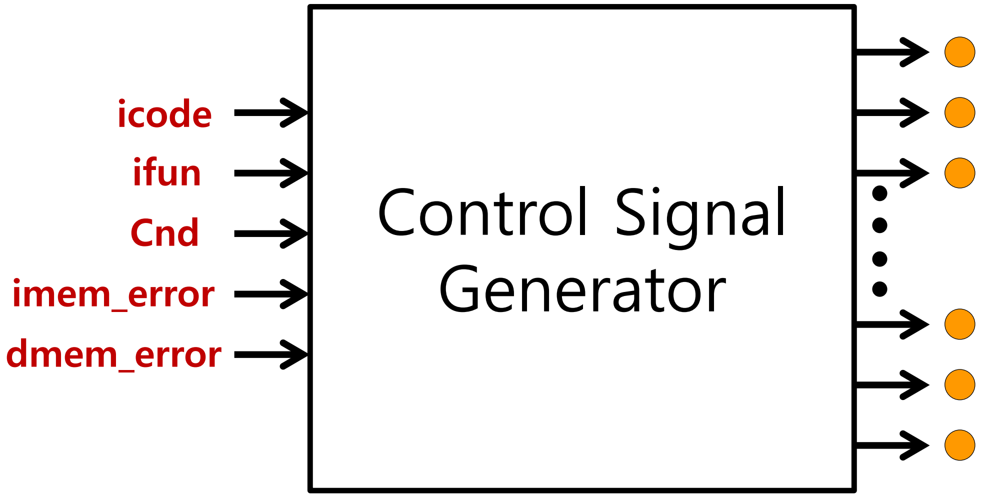
Key

- Yellow circles ●
  Control signals
  - MUX select inputs
  - Memory/Register enable signals
  - ALU functions
- Blue boxes:
  Machine states
- No Gray boxes:
  control signals explicitly represented



Control Signal Generation

- Control Signals
  - MUX select inputs
  - Memory/Register enable
  - ALU functions

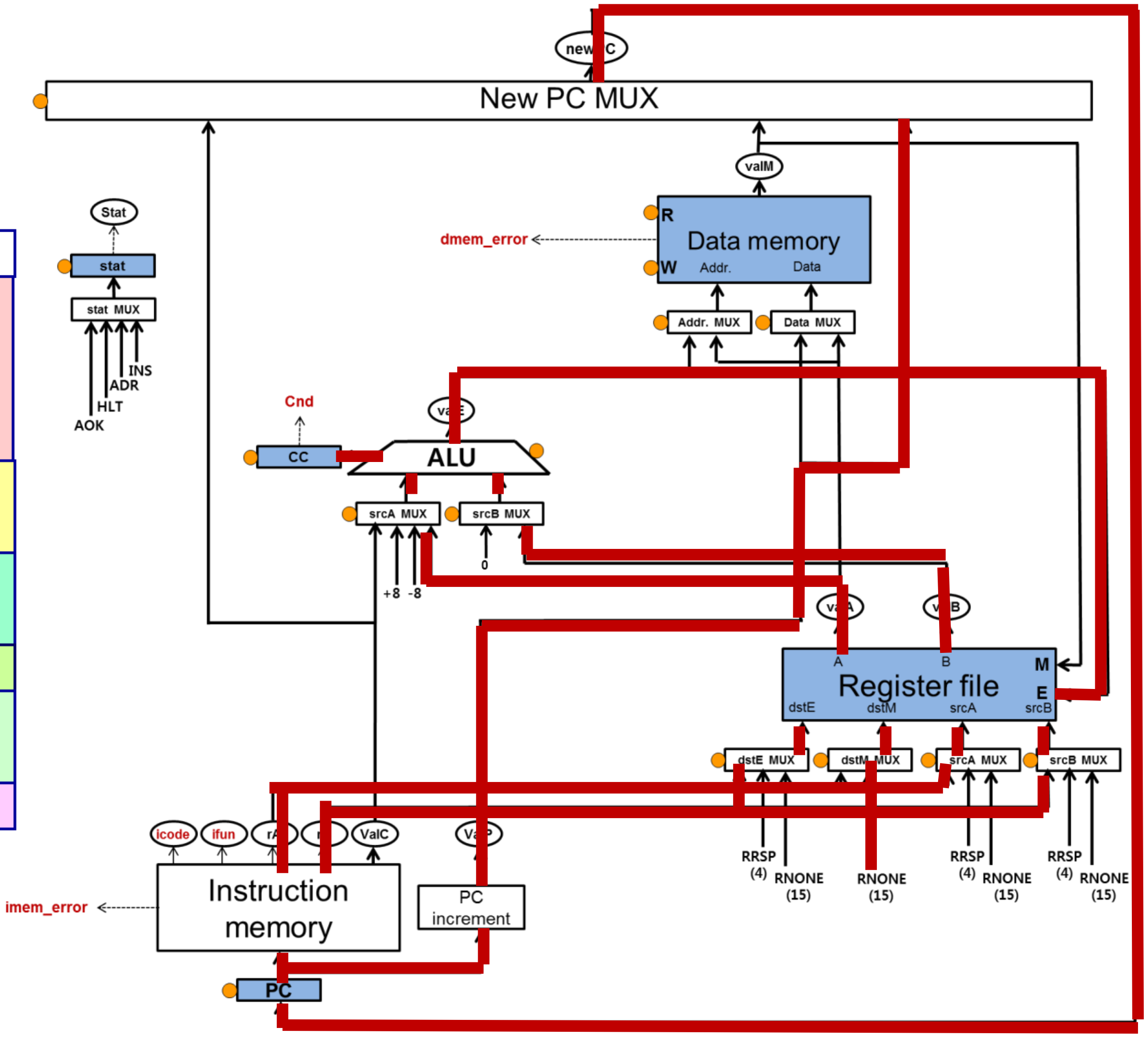


Opq Instruction

	OPq rA, rB
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2

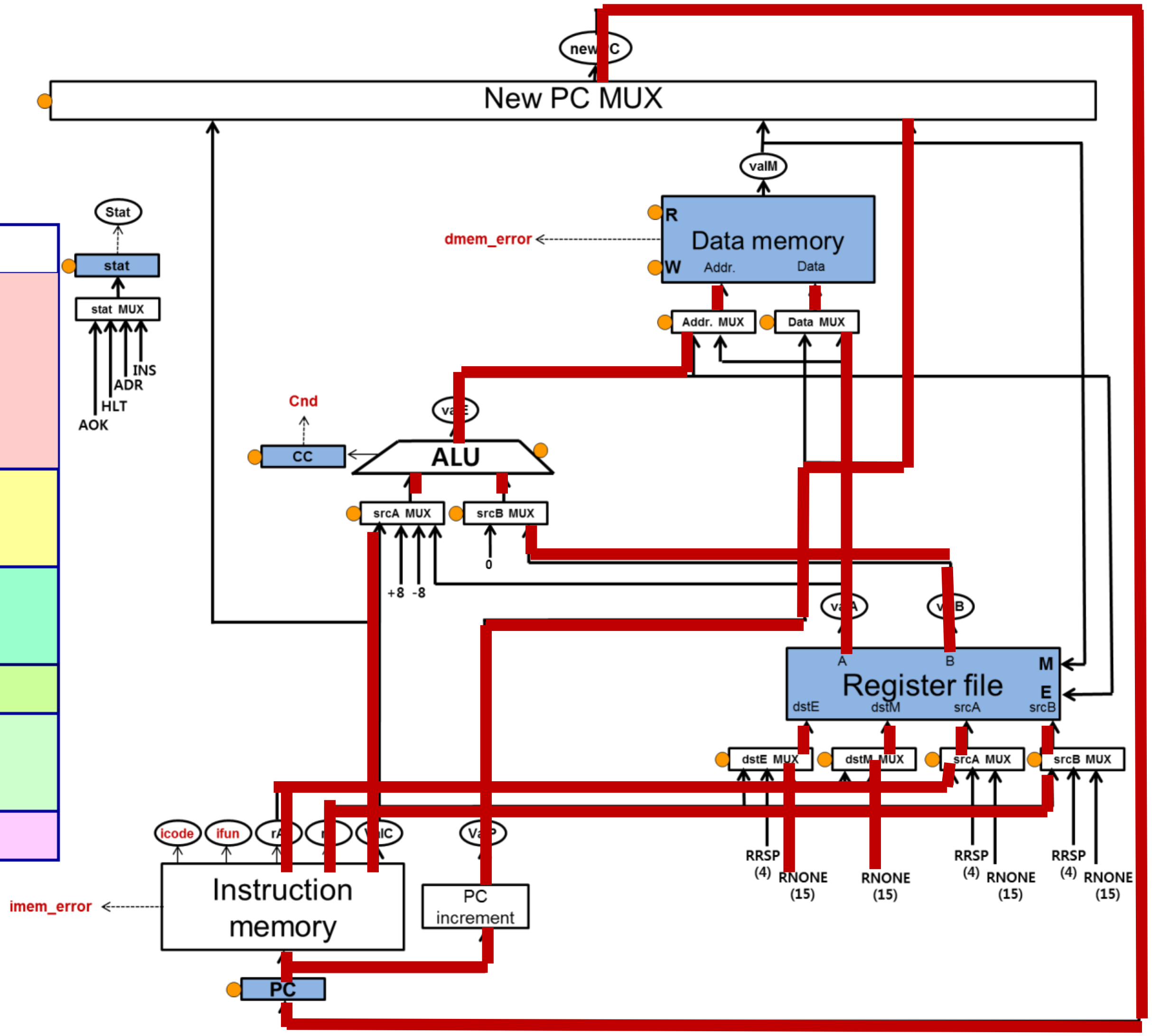
Decode	valA ← R[rA]
	valB ← R[rB]

Execute	valE ← valB OP valA Set CC
Memory	
Write back	R[rB] ← valE
PC update	PC ← valP



rmmovq Instruction

	`rmmovq rA, D(rB)`
Fetch	**icode:ifun ← M_1[PC]** **rA:rB ← M_1[PC+1]** **valC ← M_8[PC+2]** **valP ← PC+10**
Decode	**valA ← R[rA]** **valB ← R[rB]**
Execute	**valE ← valB + valC**
Memory	**M_8[valE] ← valA**
Write back	
PC update	**PC ← valP**

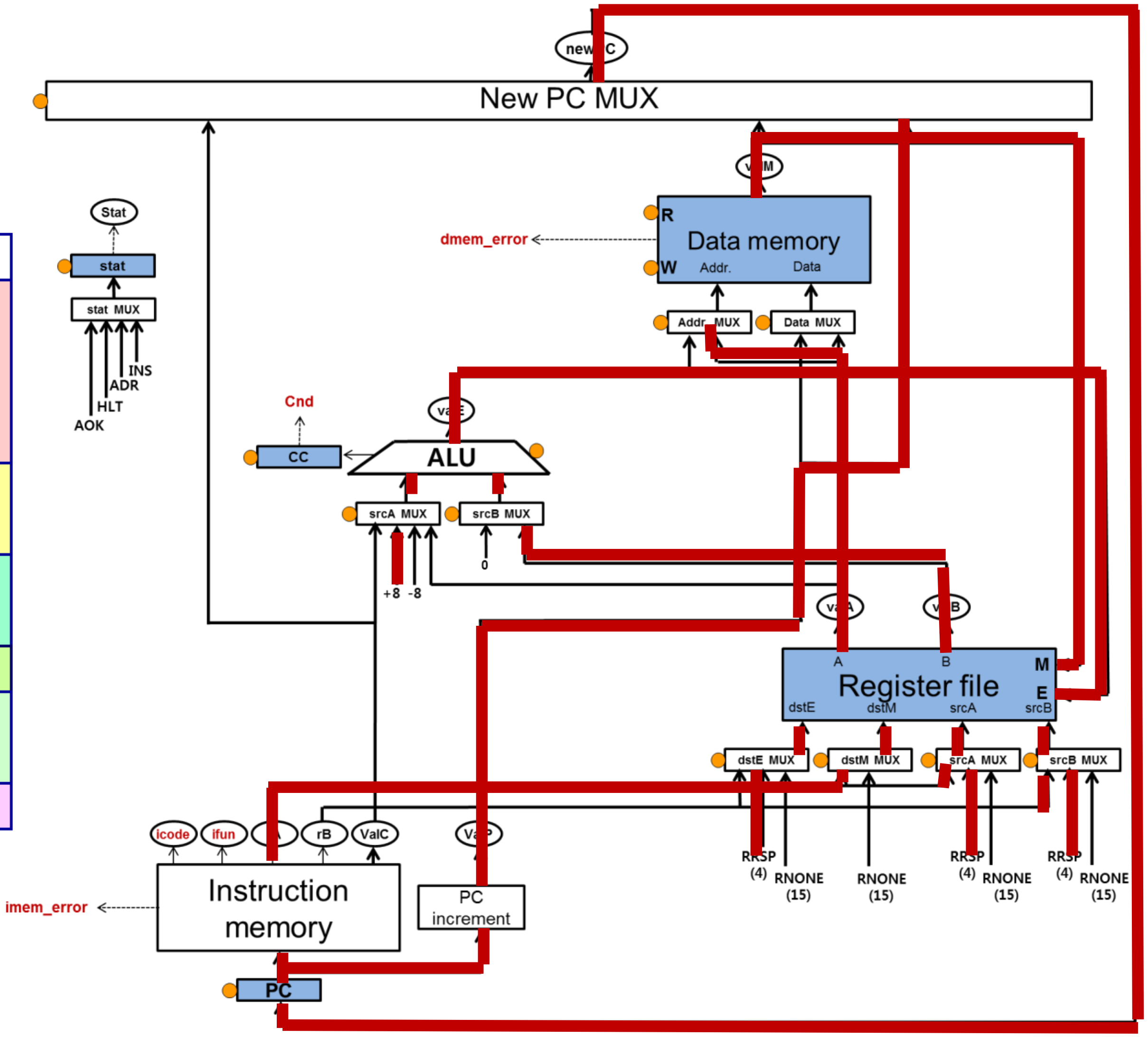


popq rA Instruction

	popq rA
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2

Decode	valA ← R[%rsp]
	valB ← R[%rsp]

Execute	valE ← valB + 8
Memory	valM ← M_8[valA]
Write back	R[%rsp] ← valE R[rA] ← valM
PC update	PC ← valP

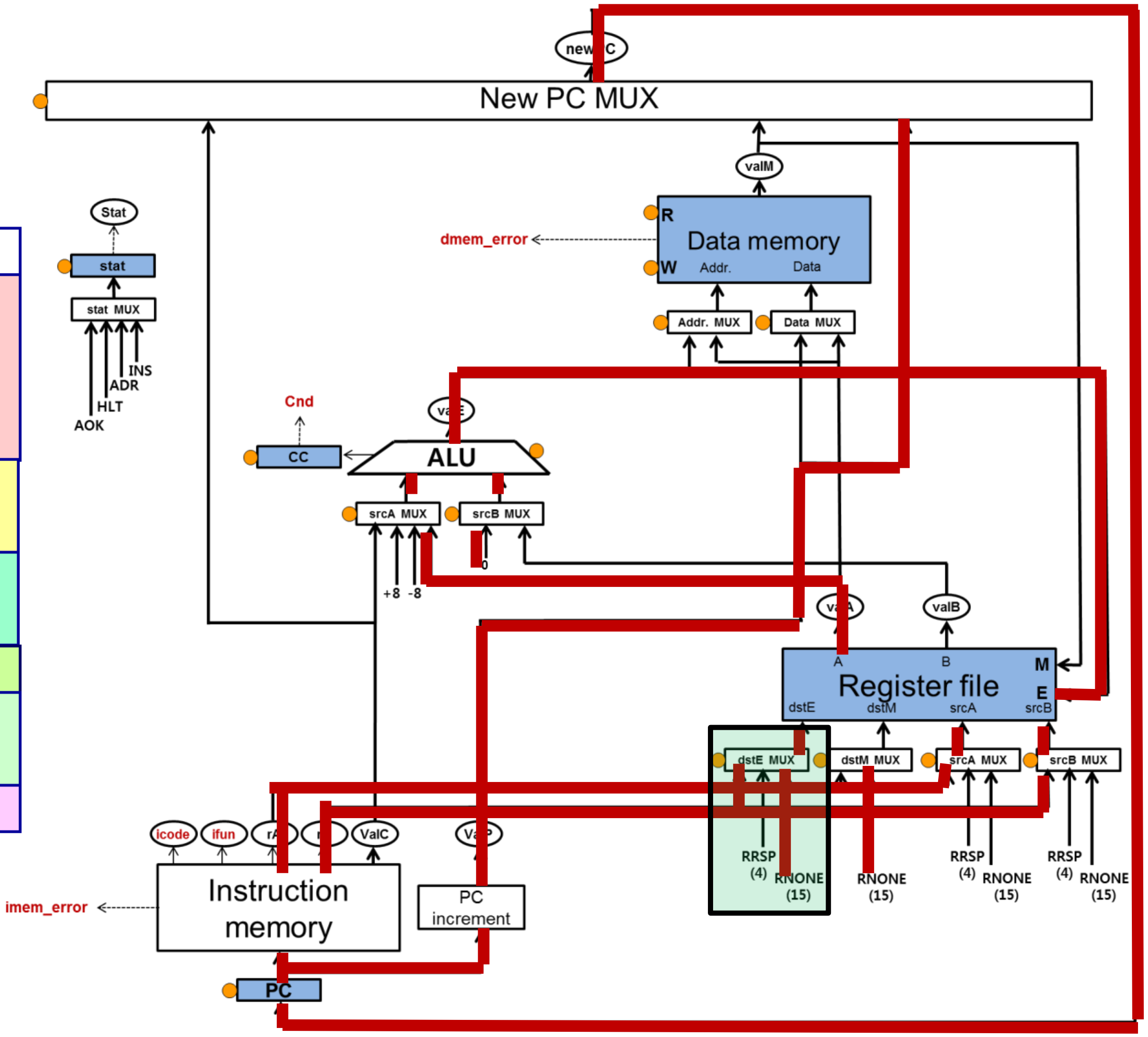


Cond. Move Instruction

	cmovXX rA, rB
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2

Decode	valA ← R[rA]
	valB ← 0

Execute	valE ← valB + valA If ! Cond(CC,ifun) rB ← 0xF
Memory	
Write back	R[rB] ← valE
PC update	PC ← valP

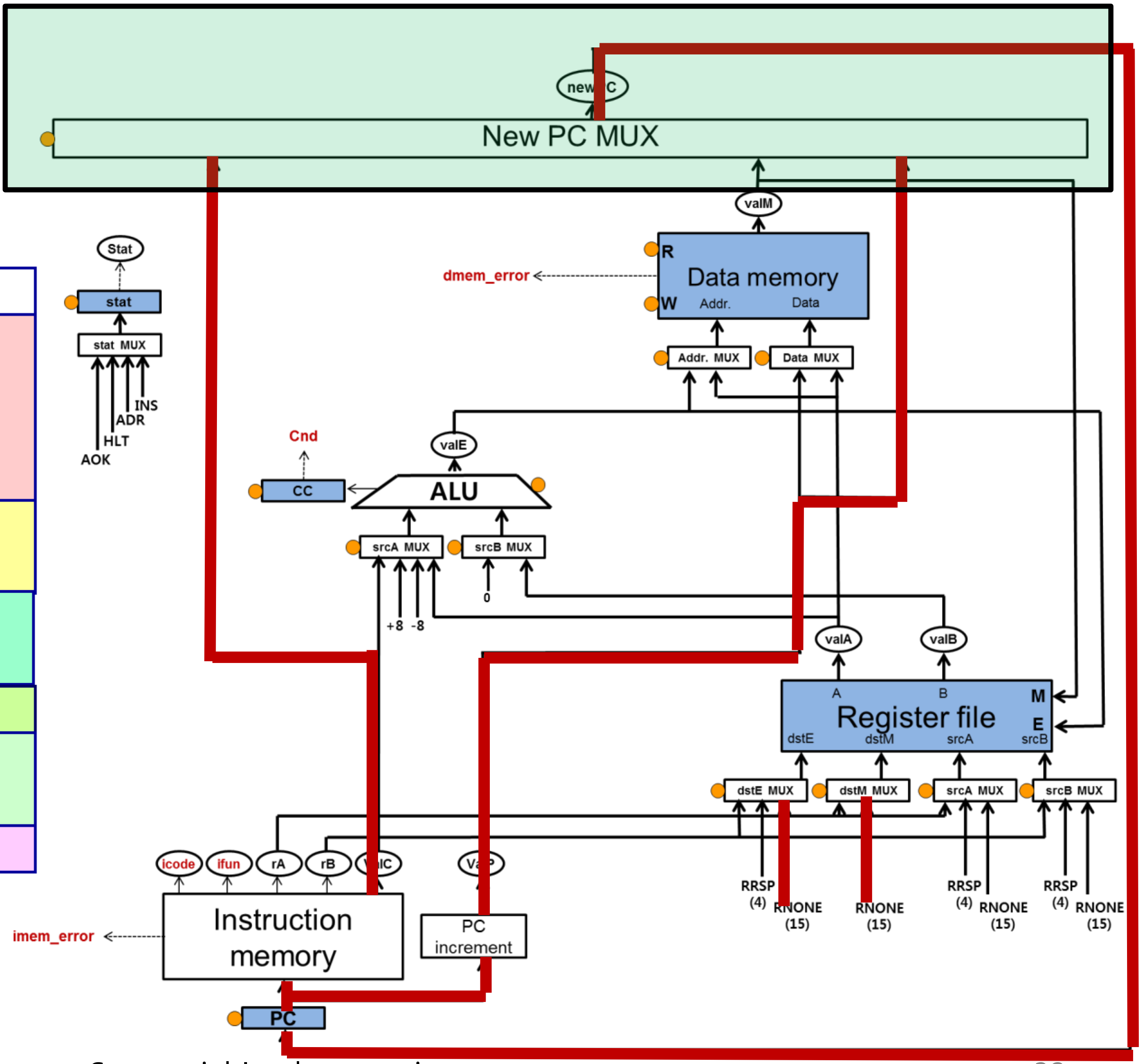


Jump Instruction

	jXX Dest

Fetch	icode:ifun ← M_1[PC]
	
	valC ← M_8[PC+1]
	valP ← PC+9

Decode	
Execute	Cnd ← Cond(CC,ifun)
Memory	
Write back	
PC update	PC ← Cnd ? valC : valP

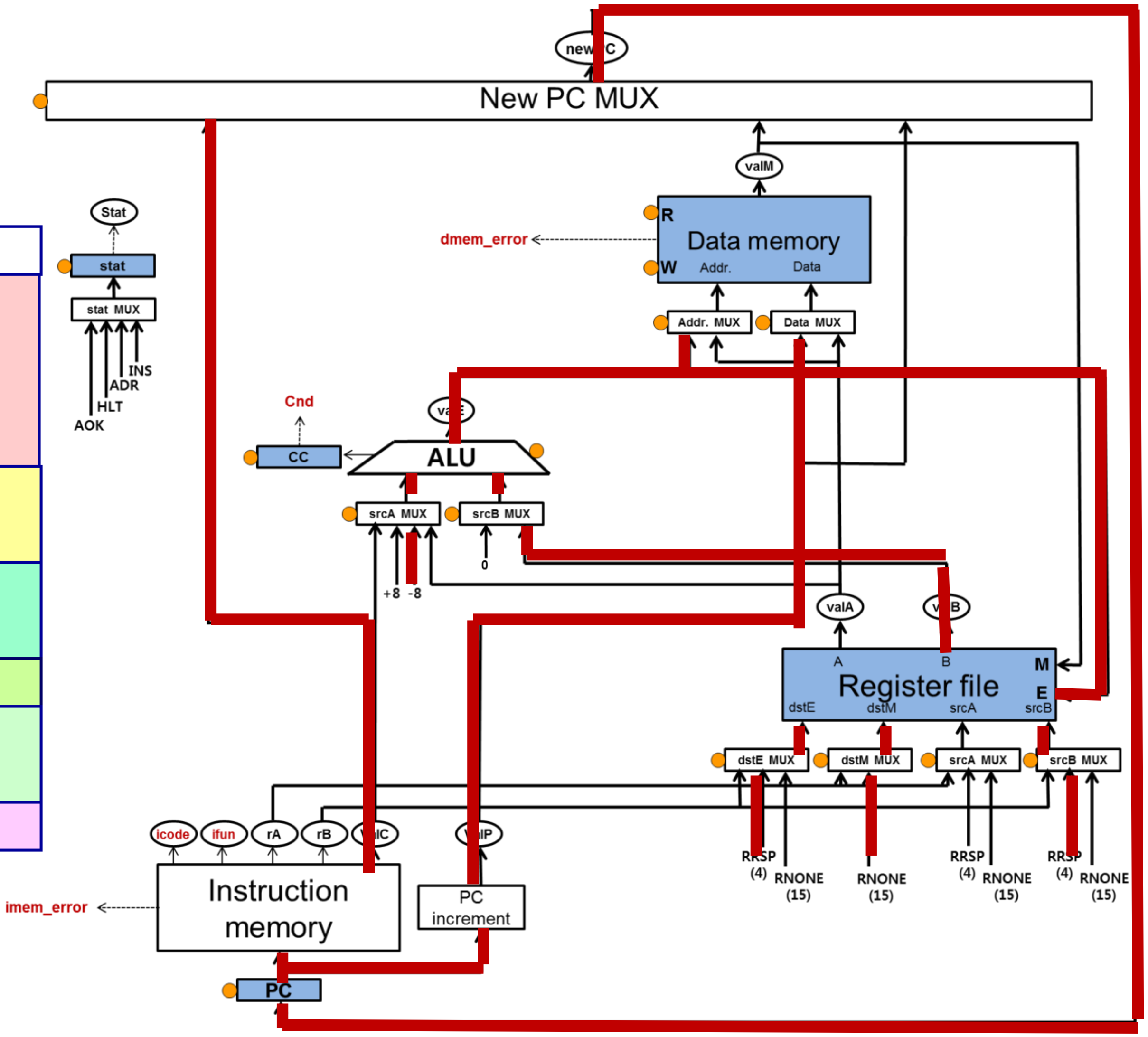


`call` Instruction

	`call` Dest

Fetch	icode:ifun ← M_1[PC]
	
	valC ← M_8[PC+1]
	valP ← PC+9

Decode	valB ← R[`%rsp`]
Execute	valE ← valB + –8
Memory	M_8[valE] ← valP
Write back	R[`%rsp`] ← valE
PC update	PC ← valC

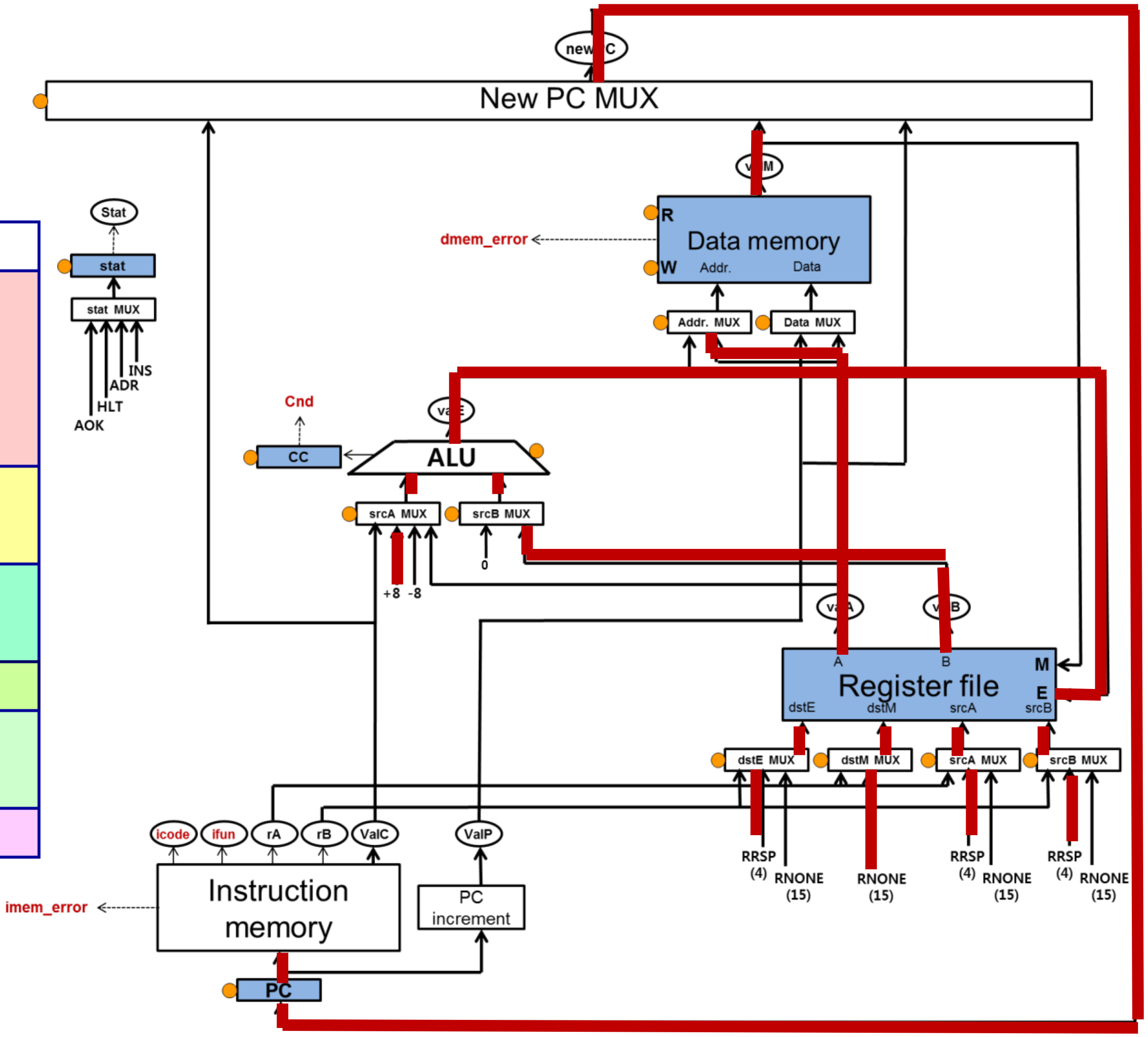


ret Instruction

	`ret`
Fetch	icode:ifun ← M_1[PC]

Decode	valA ← R[`%rsp`]
	valB ← R[`%rsp`]

Execute	valE ← valB + 8
Memory	valM ← M_8[valA]
Write back	R[`%rsp`] ← valE
PC update	PC ← valM

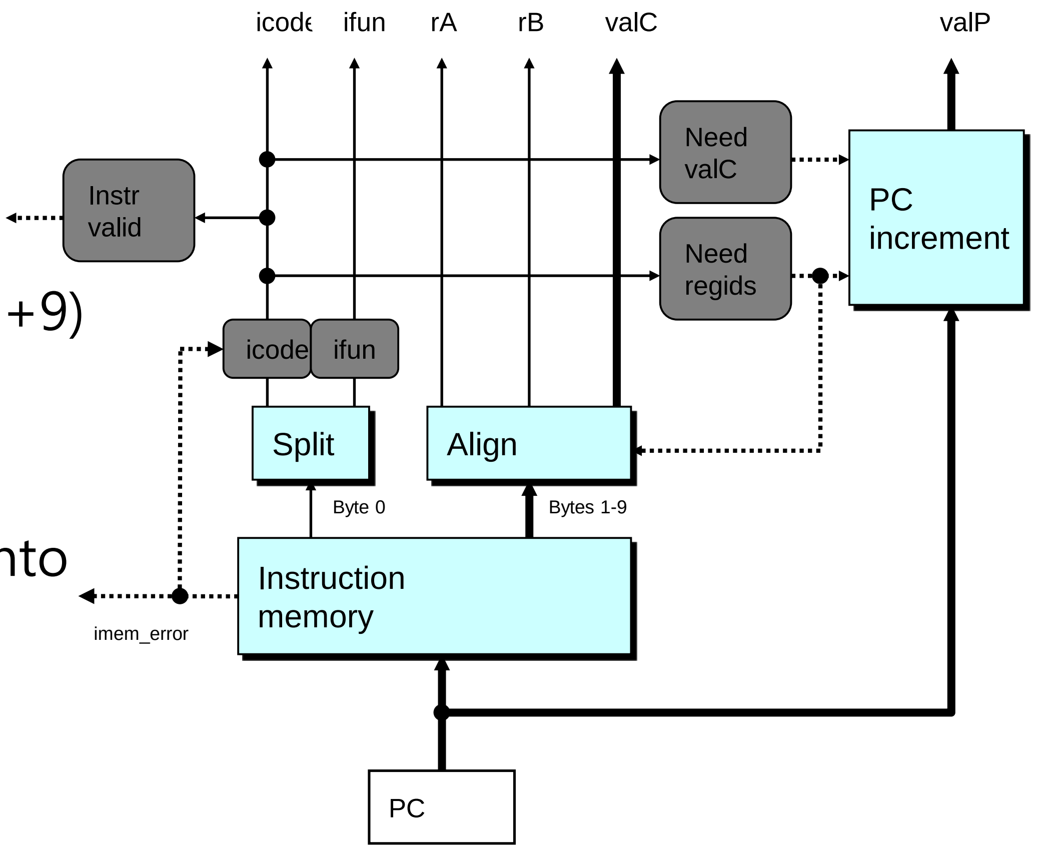


Outline

- SEQ Hardware Structure & Sequential Stages
- SEQ Stage Implementations
- SEQ Timing

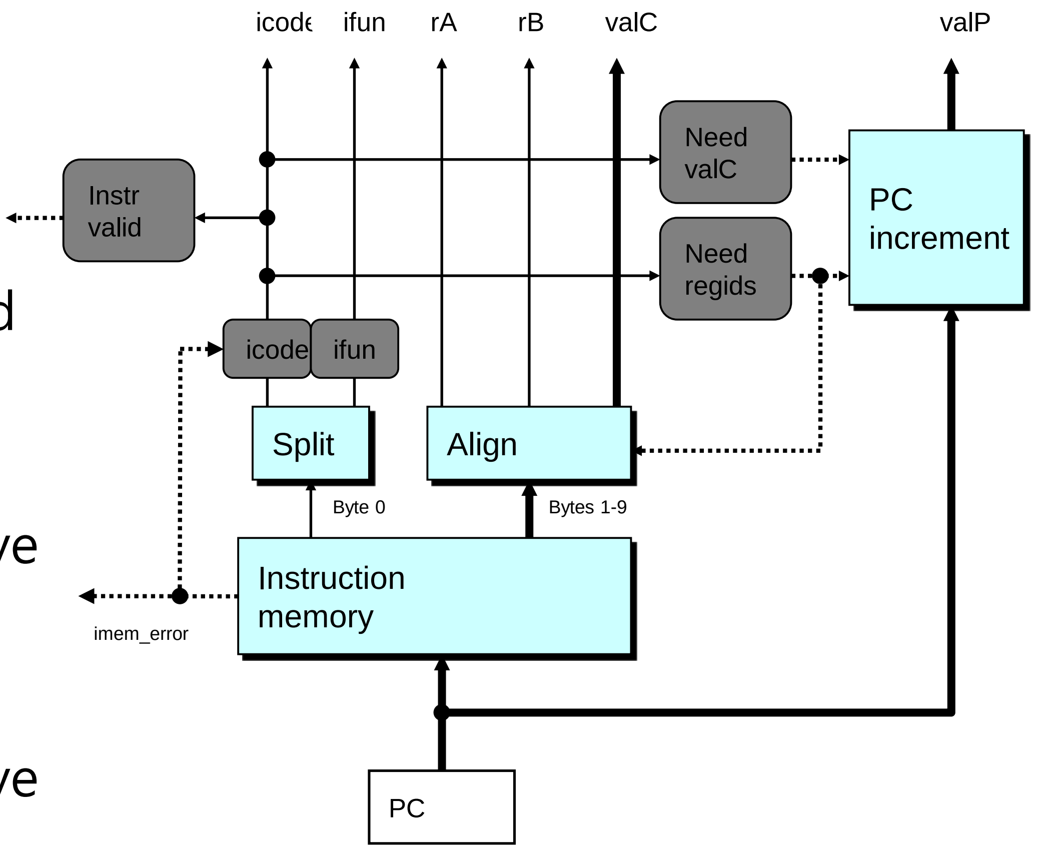
Fetch Logic: Data Path

- 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



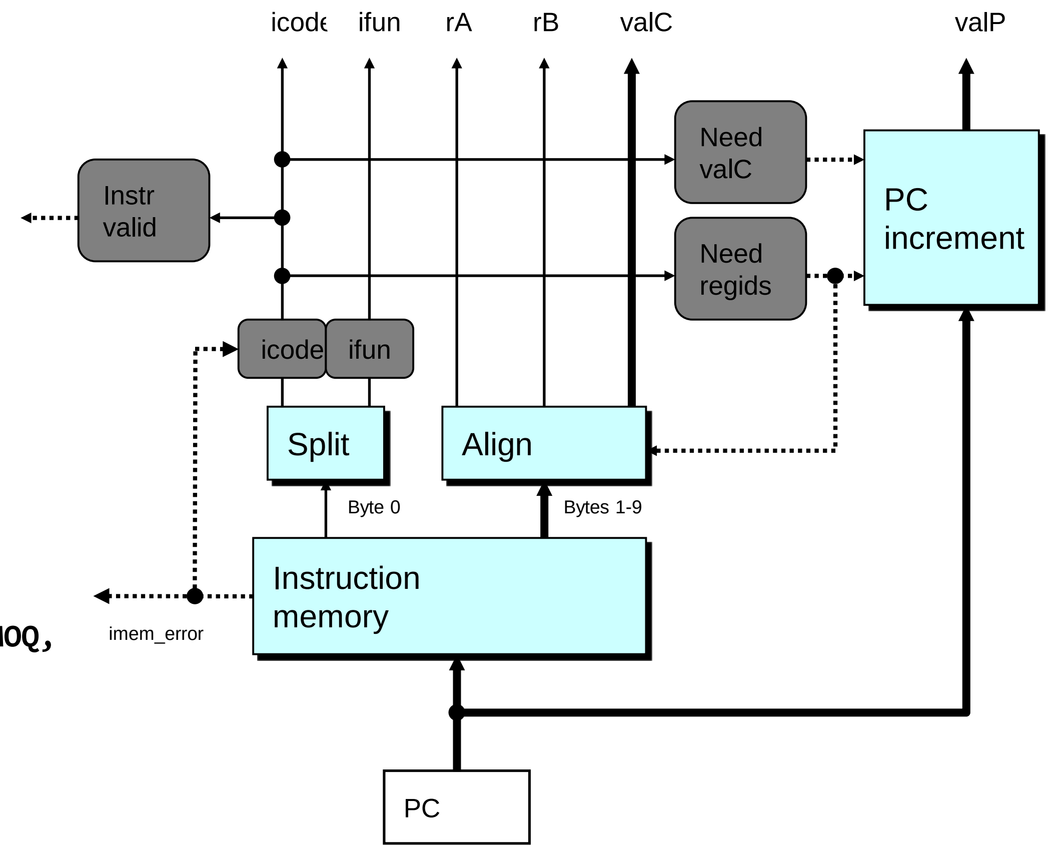
Fetch Logic: Control

- `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?

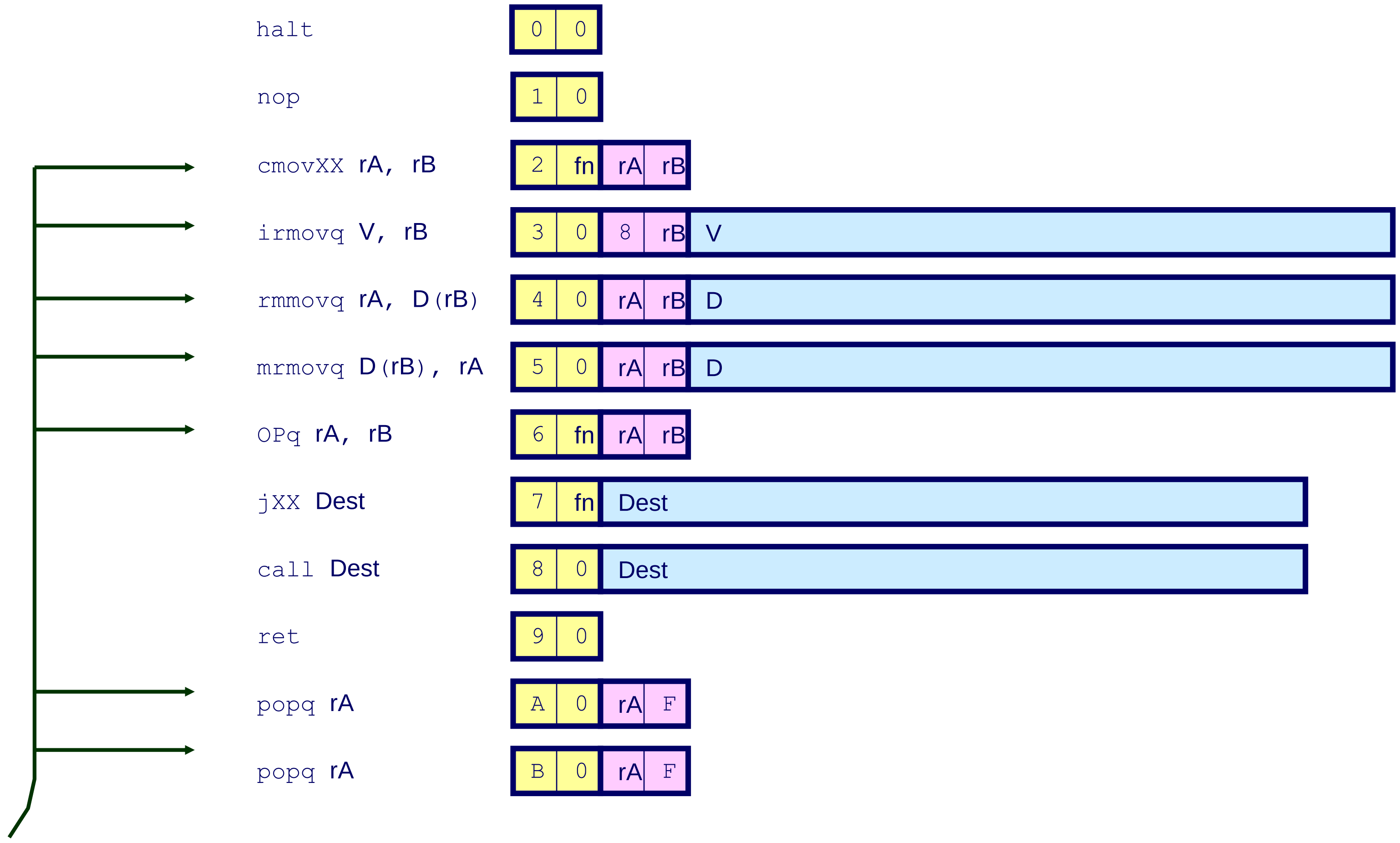


Fetch Logic: Control (in HCL)

```
int icode = [
    imem_error: INOP;
    1: imem_icode;
];

int ifun = [
    imem_error: FNONE;
    1: imem_ifun;
];

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

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

bool need_valC = icode in
    { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX,
      ICALL };
```



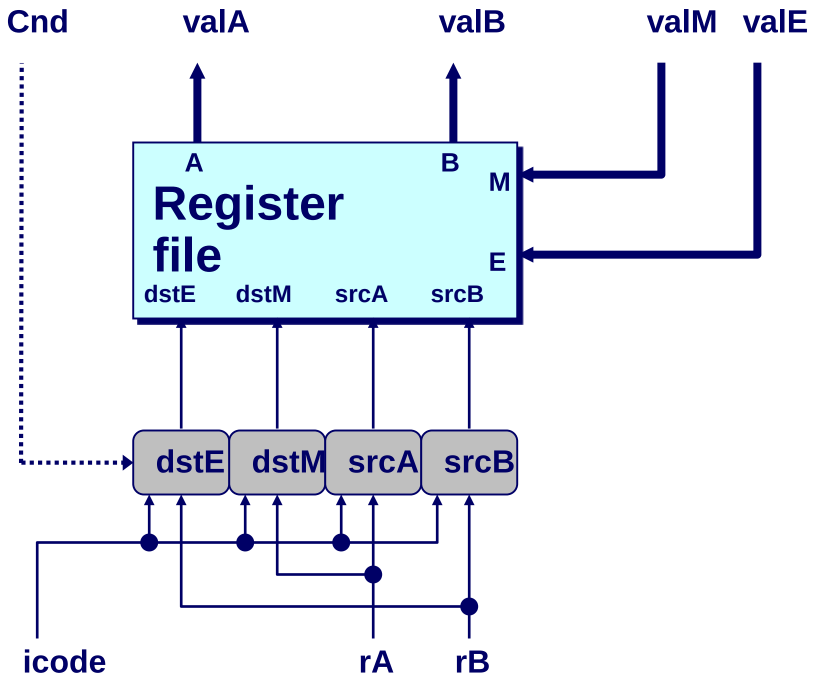
Fetch Logic: Control (in HCL)



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

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



srcA in HCL

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

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

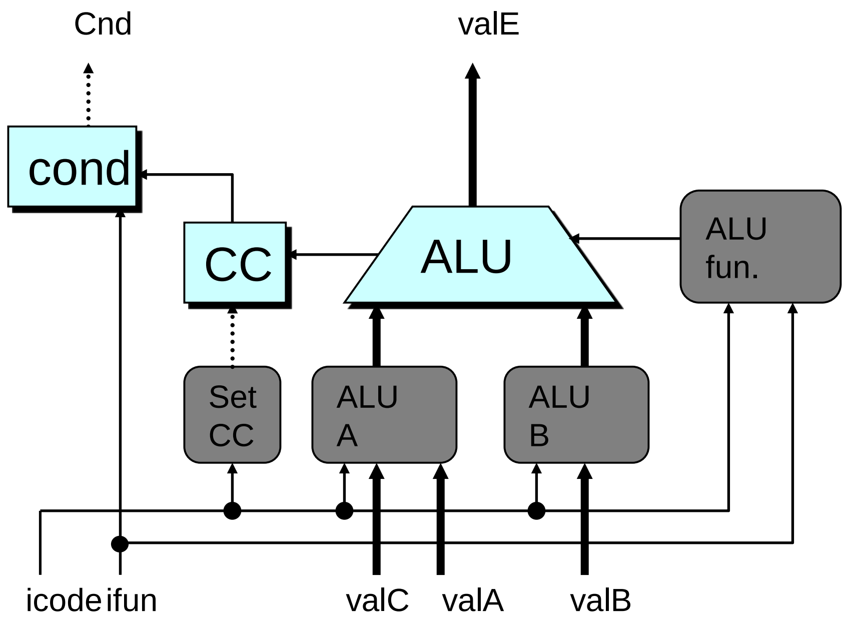
dstE in HCL

	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?



aluA Input in HCL

	OPq rA, rB	
Execute	valE ← valB OP valA	Perform ALU operation
	cmovXX rA, rB	
Execute	valE ← 0 + valA	Pass valA through ALU
	rmmovq rA, D(rB)	
Execute	valE ← valB + valC	Compute effective address
	popq rA	
Execute	valE ← valB + 8	Increment stack pointer
	jXX Dest	
Execute		No operation
	call Dest	
Execute	valE ← valB + –8	Decrement stack pointer
	ret	
Execute	valE ← 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
];
```

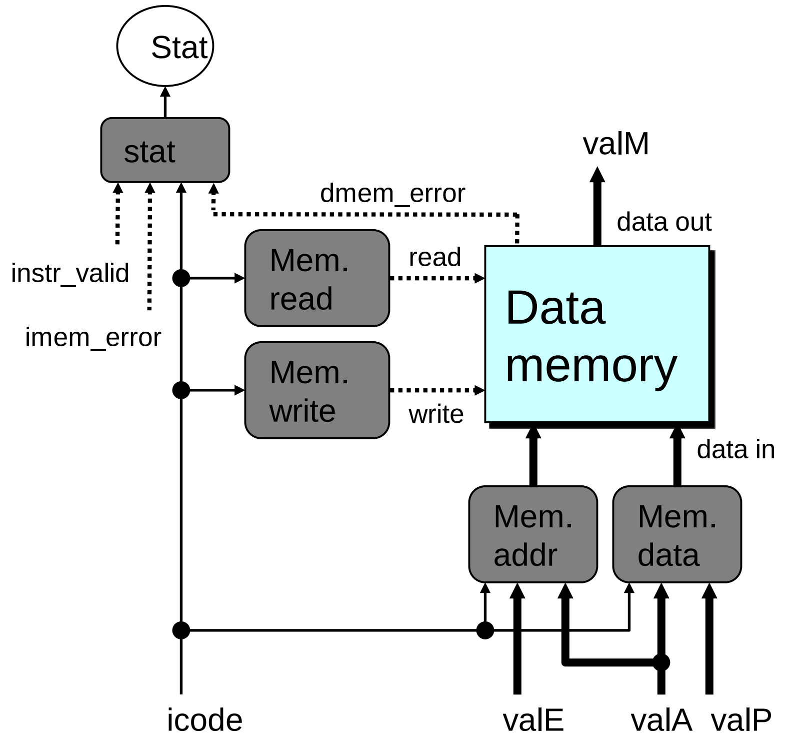
alufun Operation in HCL

	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
	`popq` rA	
Execute	valE ← valB + 8	Increment stack pointer
	jXX Dest	
Execute		No operation
	`call` Dest	
Execute	valE ← valB + –8	Decrement stack pointer
	`ret`	
Execute	valE ← valB + 8	Increment stack pointer

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

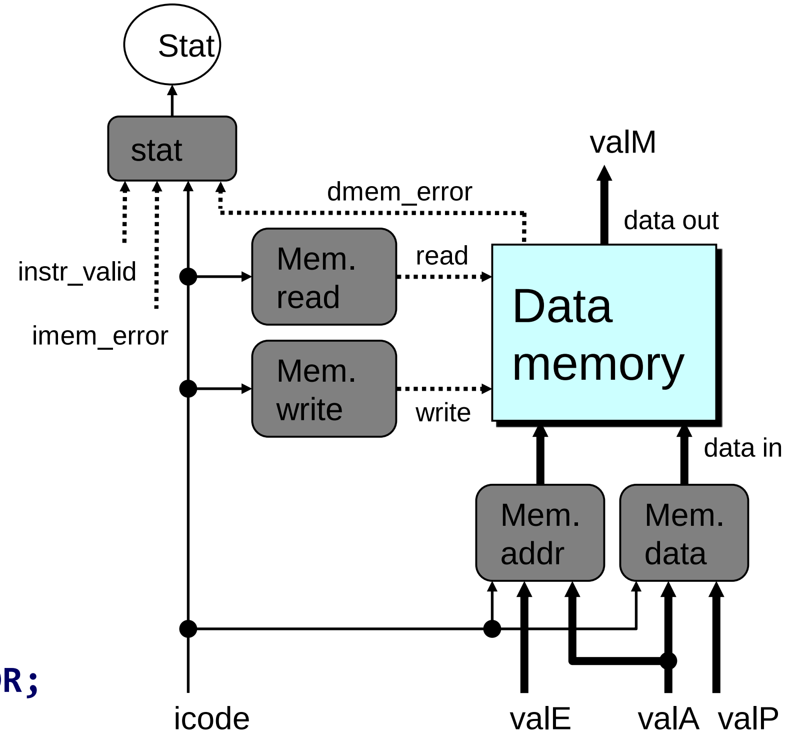
Memory Logic

- 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



stat in HCL

- 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;
];
```

mem_addr in HCL

`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 { IPOPQ, IRET } : valA;
        # Other instructions don't need address
];
```

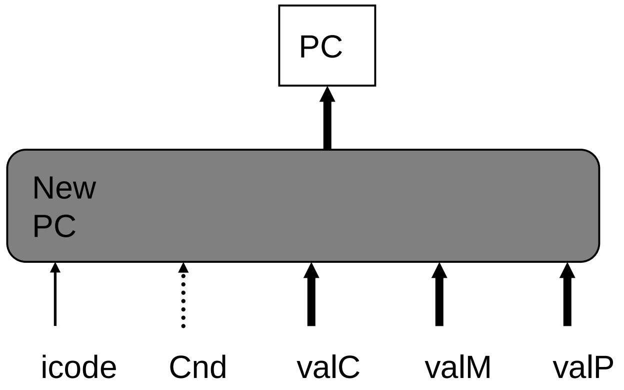
mem_read in HCL

`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, IPOPQ, IRET };
```

PC Update Logic

- New PC
  - Select next value of PC
  - One of valC, valM, or valP
  - valC: from Instruction constant
    - call, jXX
  - valM: from Data memory
    - ret
  - valP: from next PC computed



new_pc in HCL

	OPq rA, rB	
PC update	PC ← valP	Update PC
	rmmovq rA, D(rB)	
PC update	PC ← valP	Update PC
	popq 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;
];
```

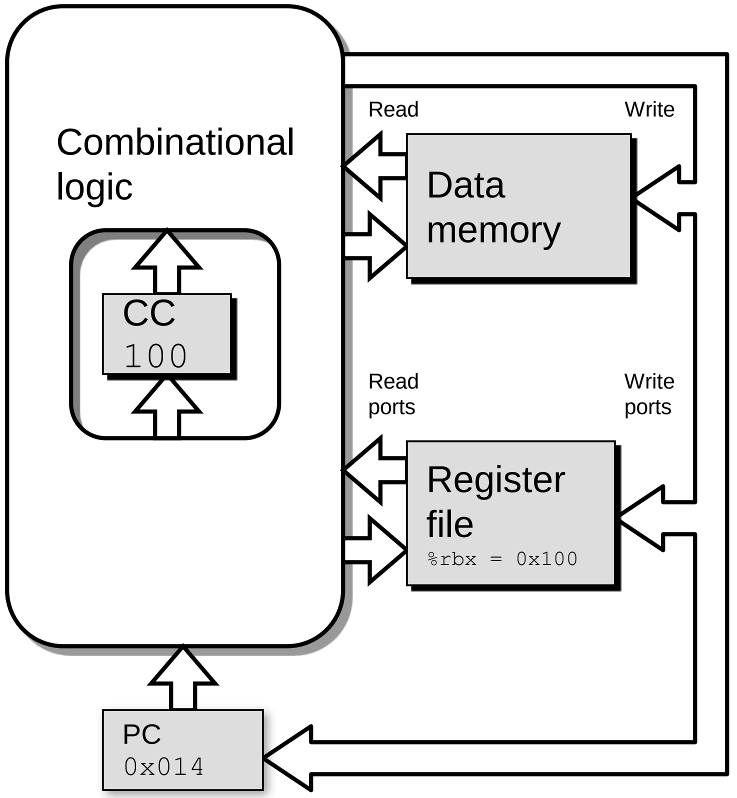
Outline

- SEQ Hardware Structure & Sequential Stages
- SEQ Stage Implementations
- SEQ Timing

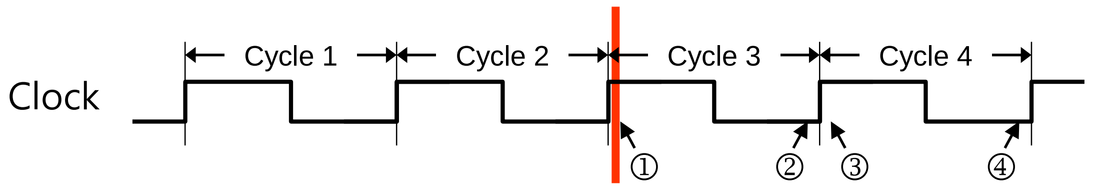
SEQ Operation

- 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

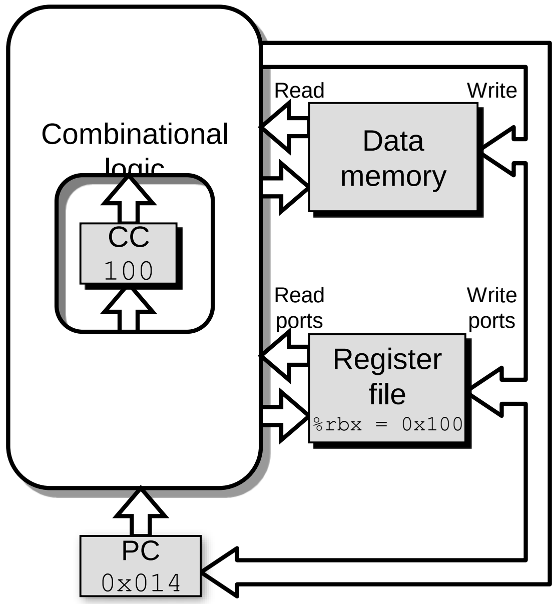


SEQ Operation #1

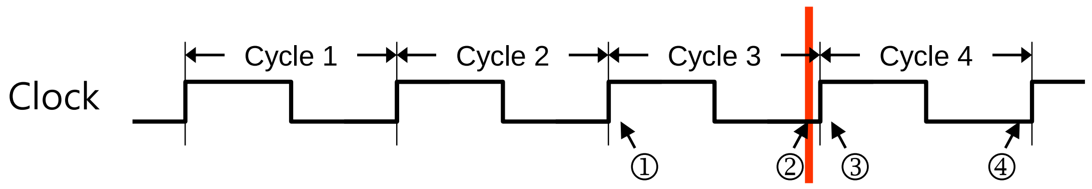


Cycle 1:	`0x000:    irmovq $0x100,%rbx   # %rbx <-- 0x100`
Cycle 2:	`0x00a:    irmovq $0x200,%rdx   # %rdx <-- 0x200`
Cycle 3:	`0x014:    addq %rdx,%rbx       # %rbx <-- 0x300 CC <-- 000`
Cycle 4:	`0x016:    je dest              # Not taken`
Cycle 5:	`0x01f:    rmmovq %rbx,0(%rdx)  # M[0x200] <-- 0x300`

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

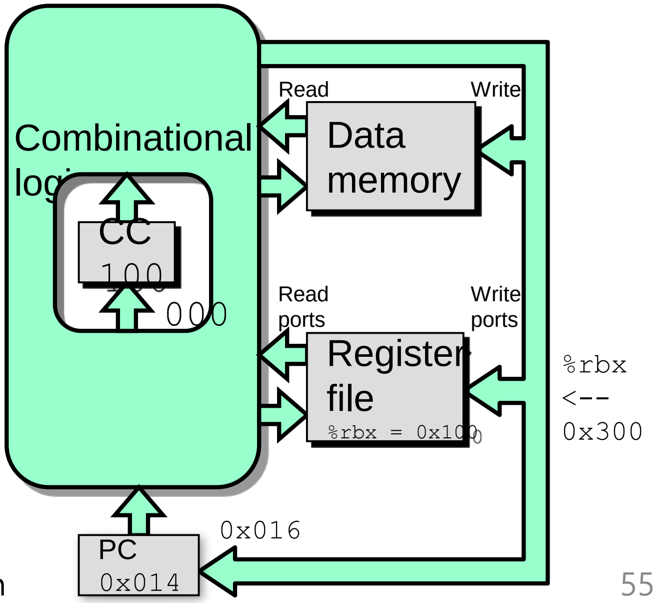


SEQ Operation #2

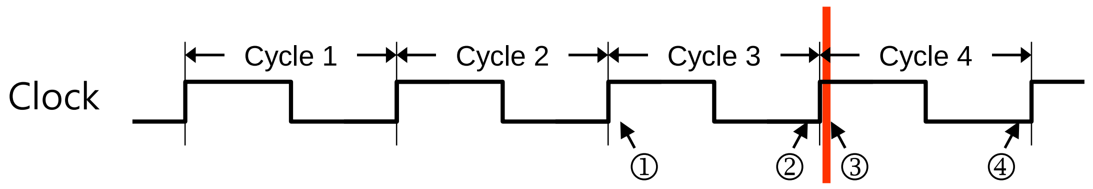


Cycle 1:	`0x000:    irmovq $0x100,%rbx   # %rbx <-- 0x100`
Cycle 2:	`0x00a:    irmovq $0x200,%rdx   # %rdx <-- 0x200`
Cycle 3:	`0x014:    addq %rdx,%rbx       # %rbx <-- 0x300 CC <-- 000`
Cycle 4:	`0x016:    je dest              # Not taken`
Cycle 5:	`0x01f:    rmmovq %rbx,0(%rdx)  # M[0x200] <-- 0x300`

- State set according to second **`irmovq`** instruction
- Combinational logic generates results for **`addq`** instruction

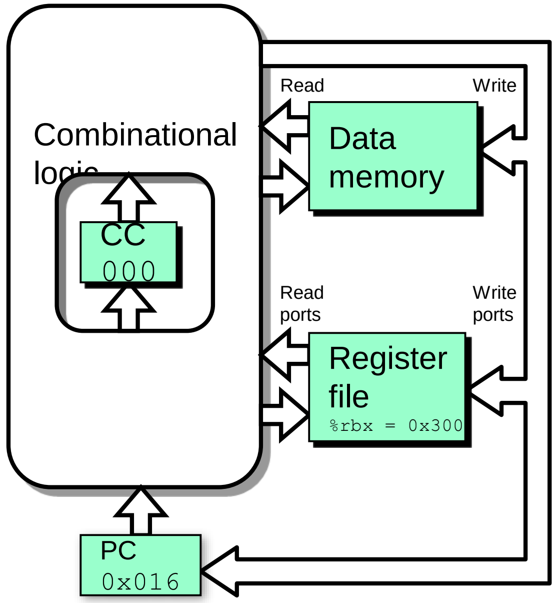


SEQ Operation #3

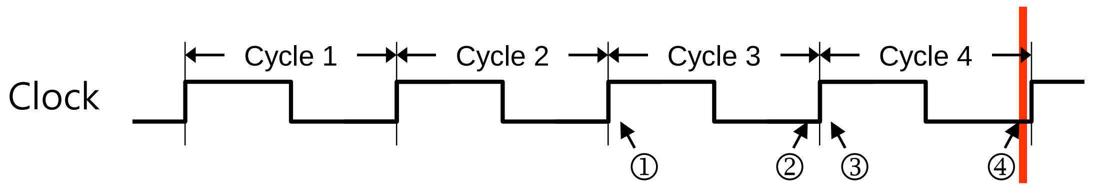


Cycle 1:	0x000: irmovq $0x100,%rbx # %rbx <-- 0x100
Cycle 2:	0x00a: irmovq $0x200,%rdx # %rdx <-- 0x200
Cycle 3:	0x014: addq %rdx,%rbx # %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016: je dest # Not taken
Cycle 5:	0x01f: rmmovq %rbx,0(%rdx) # M[0x200] <-- 0x300

- State set according to addq instruction
- Combinational logic starting to react to state changes

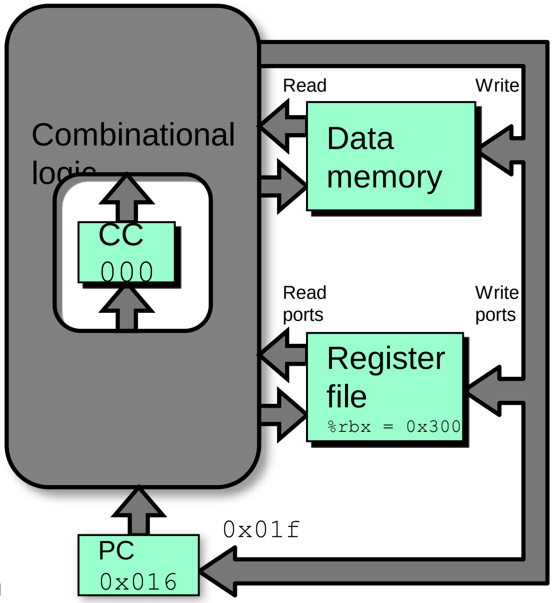


SEQ Operation #4



```
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 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

Questions?