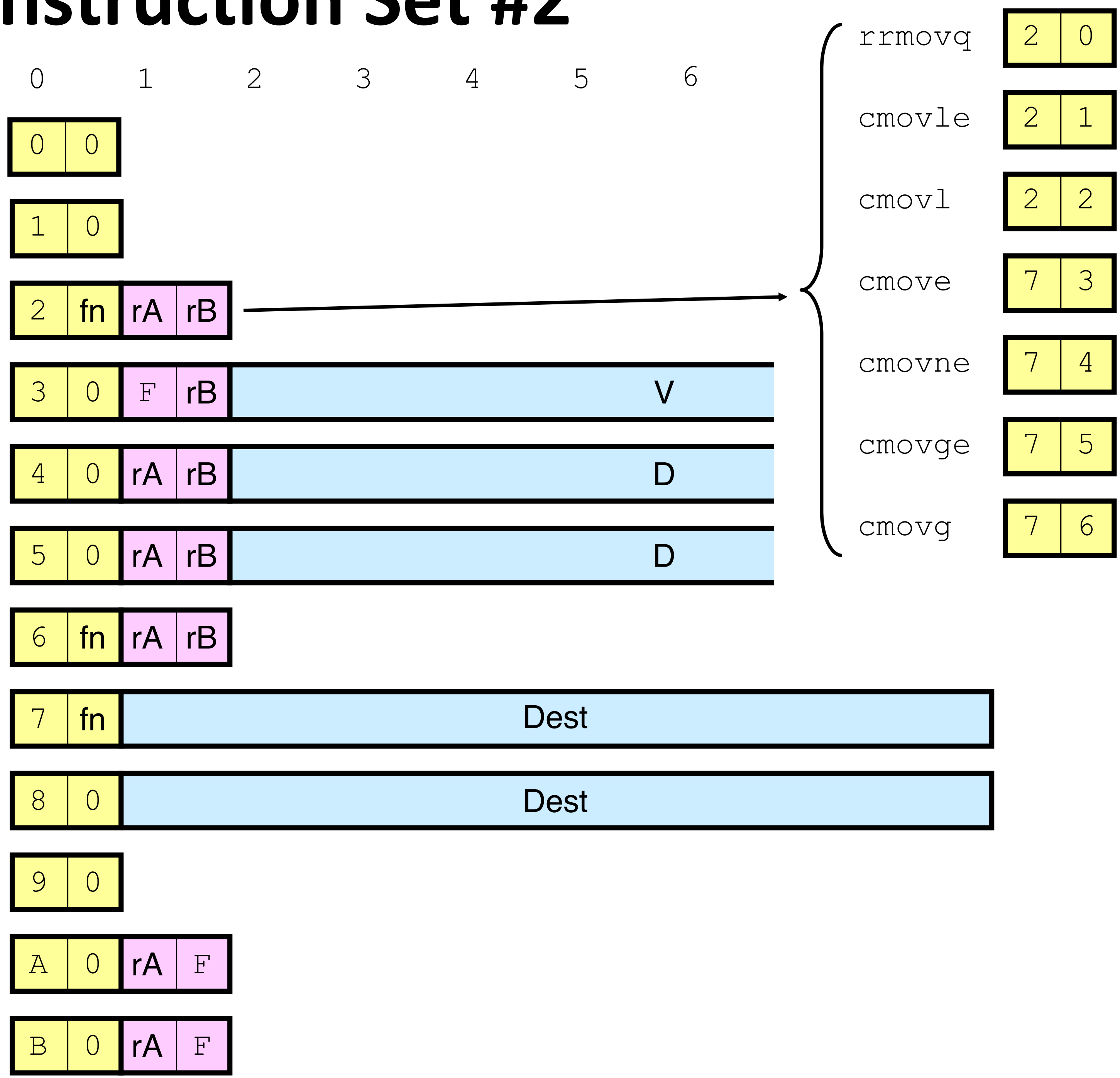
Processor Architecture: Sequential Implementation

Y86-64 Instruction Set #1

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

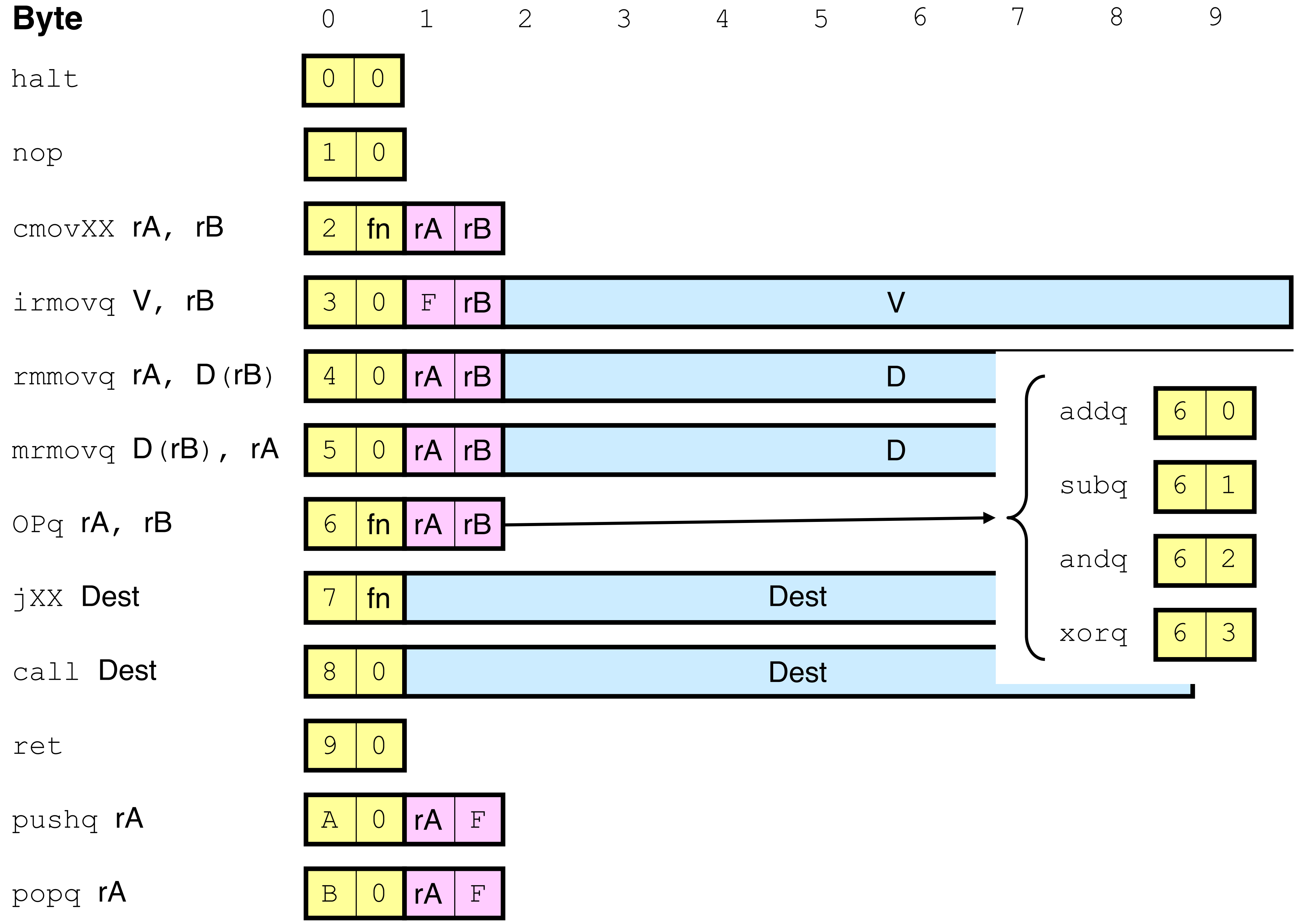
Y86-64 Instruction Set #2



Byte	0	1	2	3	4	5	6
`halt`	0 0						
`nop`	1 0						
`cmovXX` rA, rB	2 fn	rA rB					
`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					
`call` Dest	8 0	Dest					
`ret`	9 0						
`pushq` rA	A 0	rA F					
`popq` rA	B 0	rA F					

`rrmovq`	2 0
`cmovle`	2 1
`cmovl`	2 2
`cmove`	7 3
`cmovne`	7 4
`cmovge`	7 5
`cmovg`	7 6

Y86-64 Instruction Set #3



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

`addq`	6 0
`subq`	6 1
`andq`	6 2
`xorq`	6 3

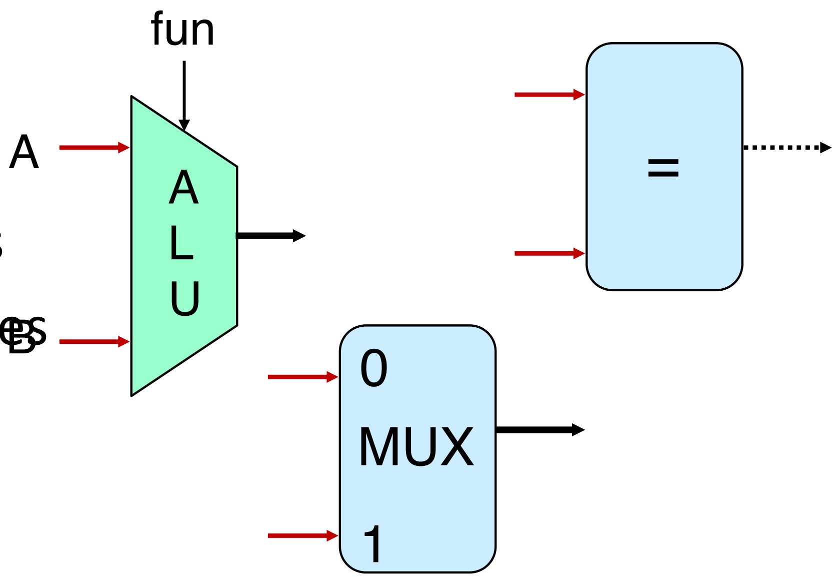
Y86-64 Instruction Set #4

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

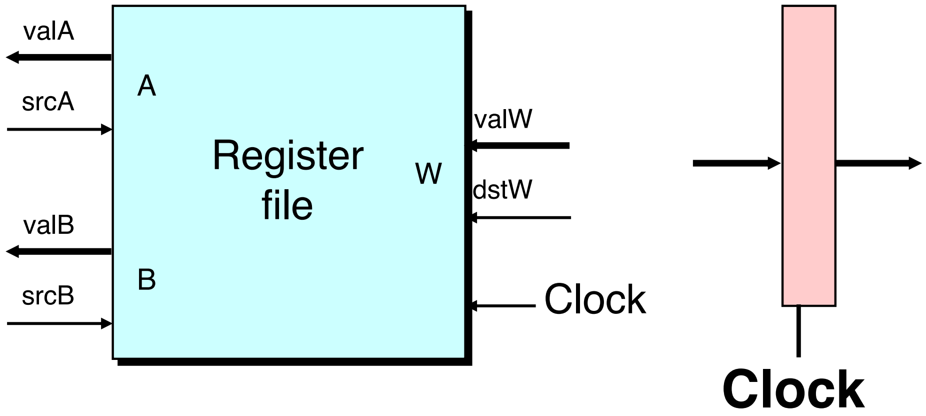
`jmp`	7	0
`jle`	7	1
`jl`	7	2
`je`	7	3
`jne`	7	4
`jge`	7	5
`jg`	7	6

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



Hardware Control Language

- Very simple hardware description language
- Can only express limited aspects of hardware operation
  - Parts we want to explore and modify

- **Data Types**
  - `bool`: Boolean
    - `a`, `b`, `c`, …
  - `int`: words
    - `A`, `B`, `C`, …
    - Does not specify word size---bytes, 64-bit words, …

- **Statements**
  - `bool a =` *`bool-expr`* `;`
  - `int A =` *`int-expr`* `;`

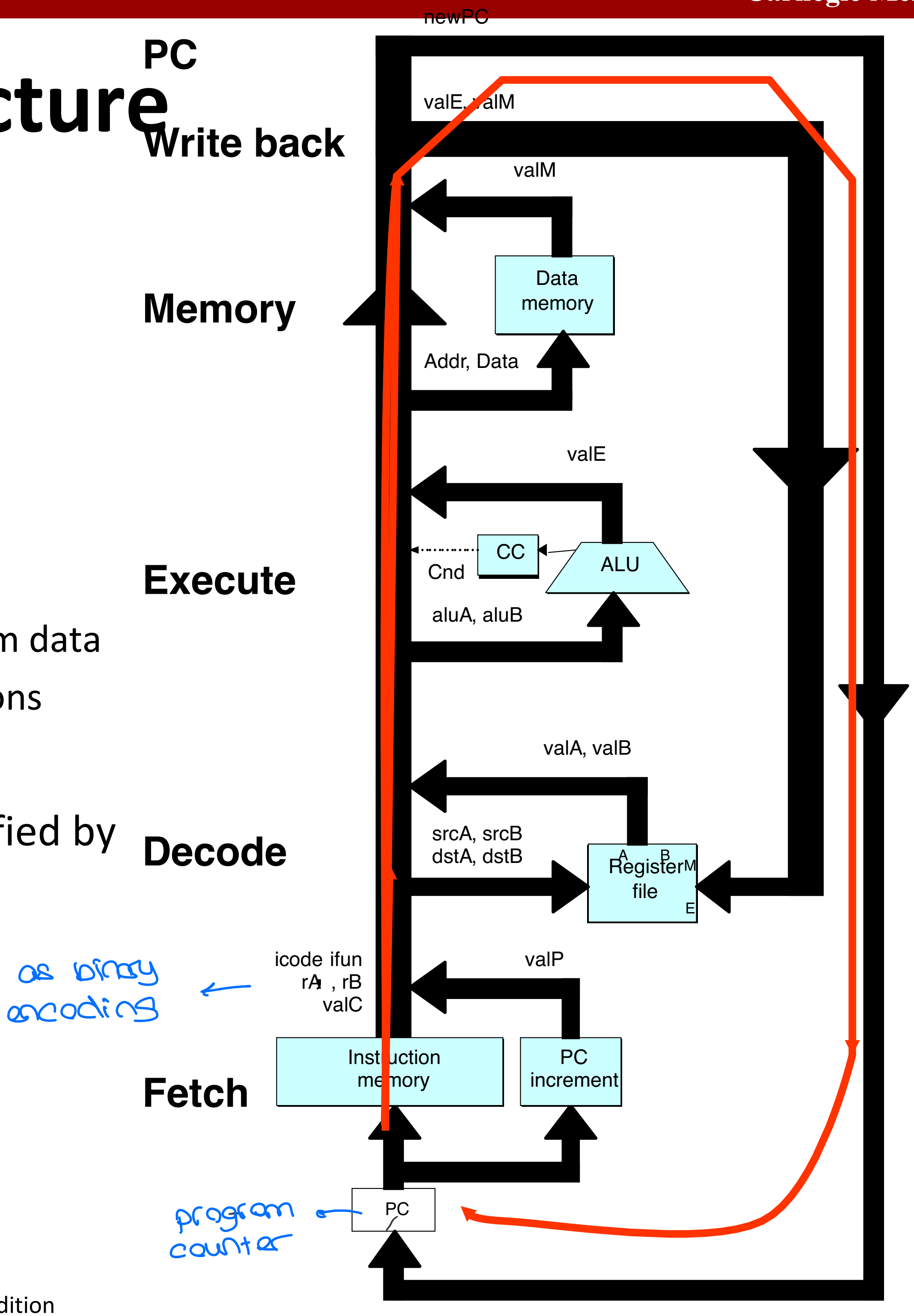
HCL Operations

  - Classify by type of value returned

- **Boolean Expressions**
  - Logic Operations
    - `a && b`, `a || b`, `!a`
  - Word Comparisons
    - `A == B`, `A != B`, `A < B`, `A <= B`, `A >= B`, `A > B`
  - Set Membership
    - `A in { B, C, D }`
      - Same as `A == B || A == C || A == D`
- **Word Expressions**
  - Case expressions
    - `[ a : A; b : B; c : C ]`
    - Evaluate test expressions `a`, `b`, `c`, … in sequence
    - Return word expression `A`, `B`, `C`, … for first successful test

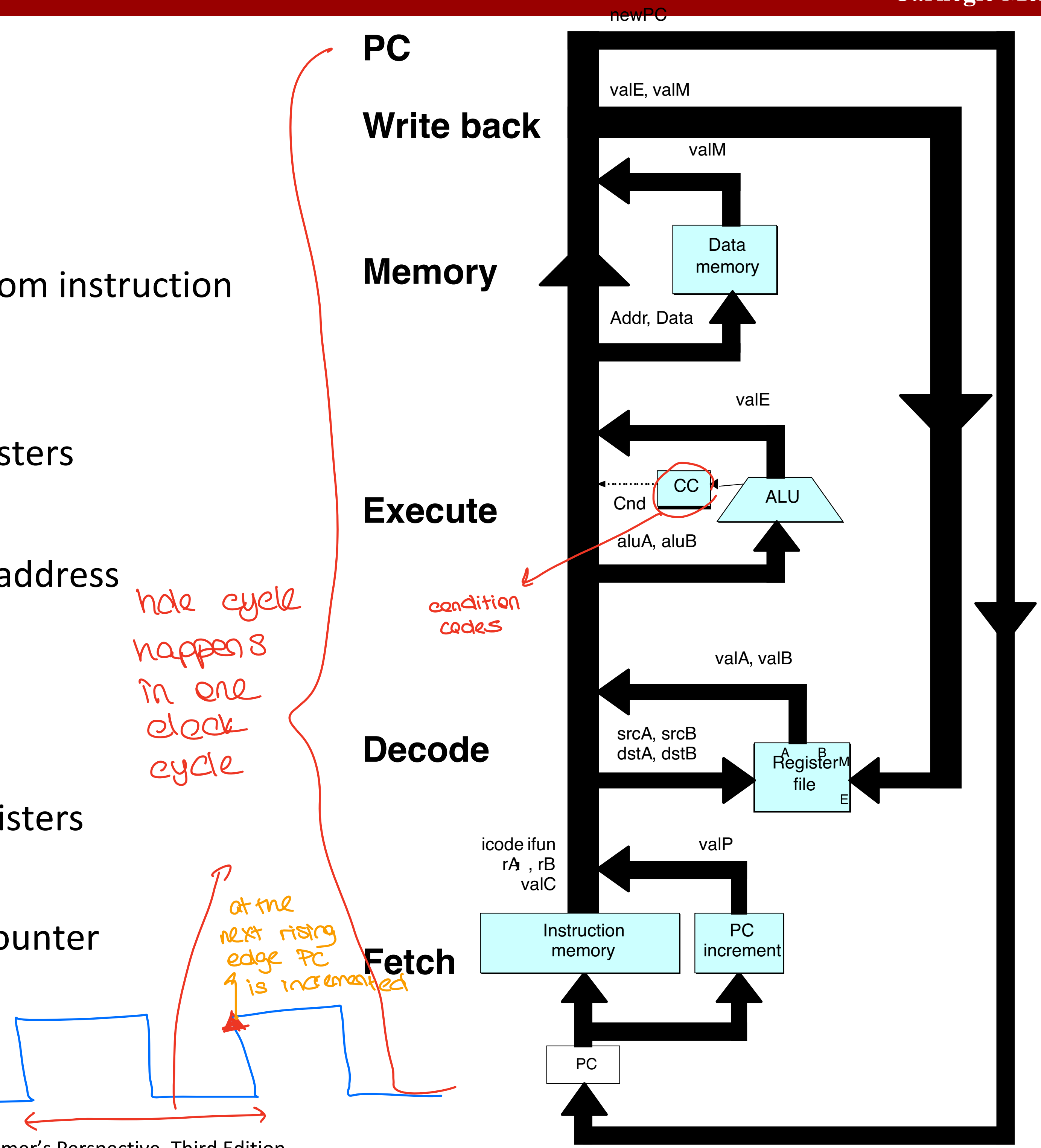
SEQ Hardware Structure

- **State**
  - Program counter register (PC)
  - Condition code register (CC)
  - Register File
  - Memories
    - Access same memory space
    - Data: for reading/writing program data
    - Instruction: for reading instructions
- **Instruction Flow**
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter



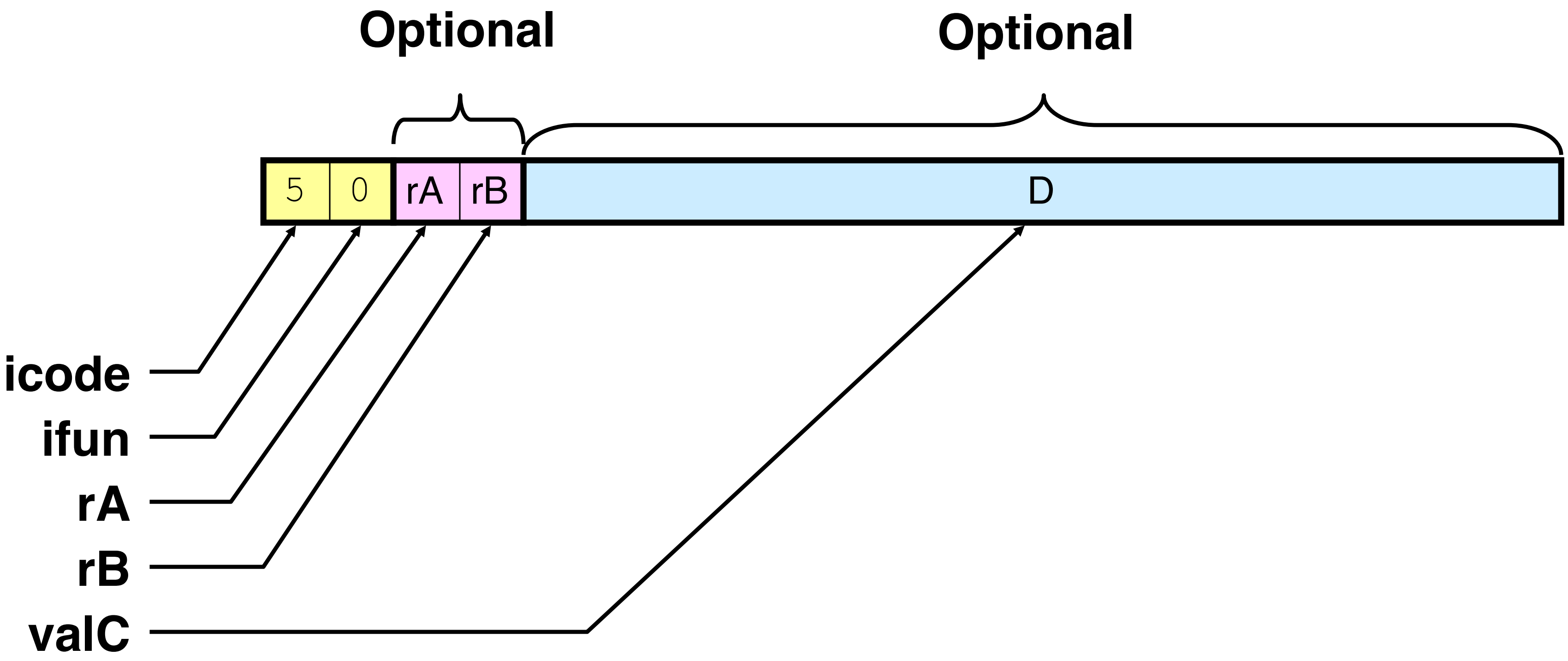
SEQ Stages

- **Fetch**
  - Read instruction from instruction memory
- **Decode**
  - Read program registers
- **Execute**
  - Compute value or address
- **Memory**
  - Read or write data
- **Write Back**
  - Write program registers
- **PC**
  - Update program counter



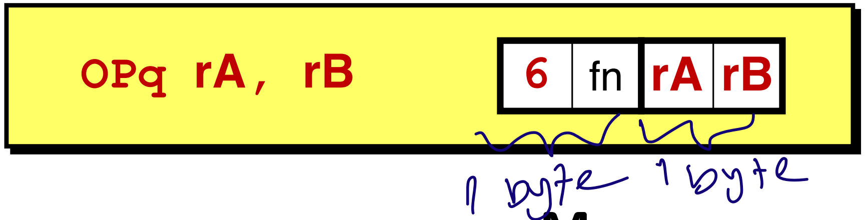
How can we implement instructions so that the whole process happens in two clock cycle

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 rB
- **PC Update**
  - Increment PC by 2

add || and || subtract ||...

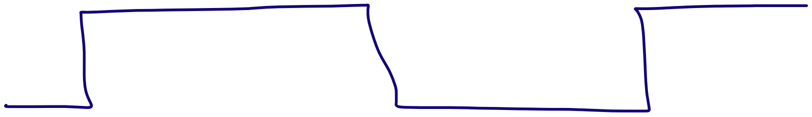
Stage Computation: Arith/Log. Ops

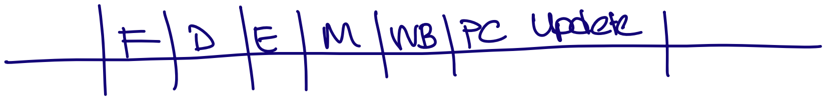
	OPq rA, rB instruction memory
Fetch they are just wires, they don't hold any value in the mean	icode:ifun ← $M_1[PC]$ 1st 4 bytes 2nd 4 bytes rA:rB ← $M_1[PC+1]$ 4 bit ↳ 4 bit valP ← PC+2

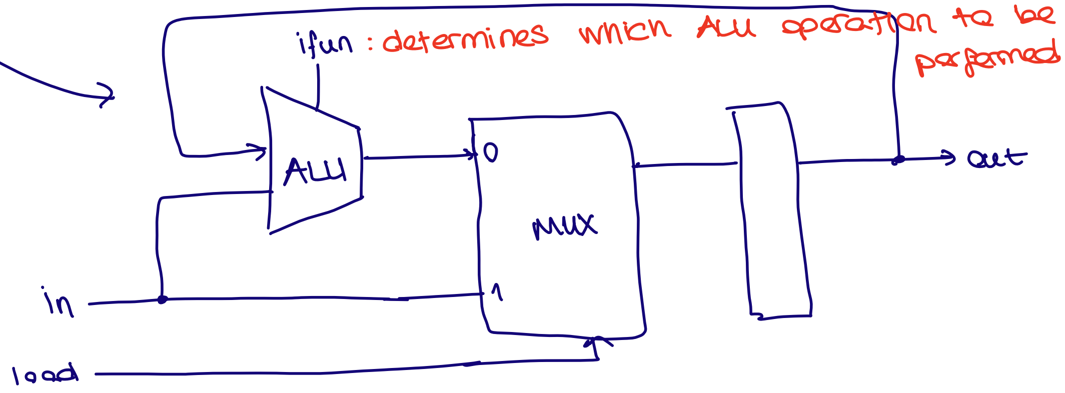
Decode	valA ← R[rA]
	valB ← R[rB]

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

we don't update anything here

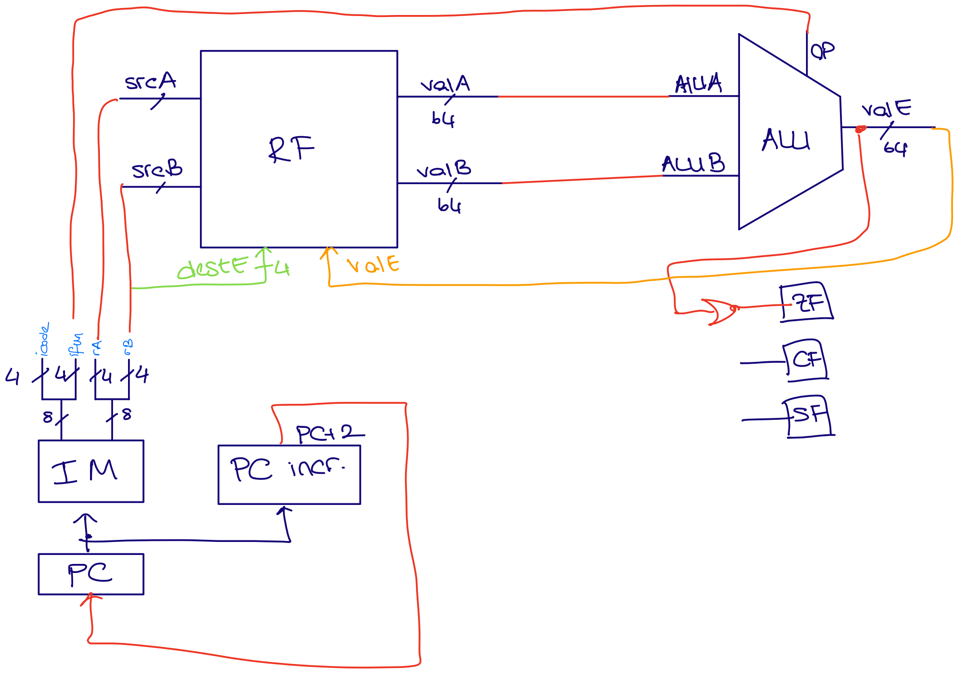


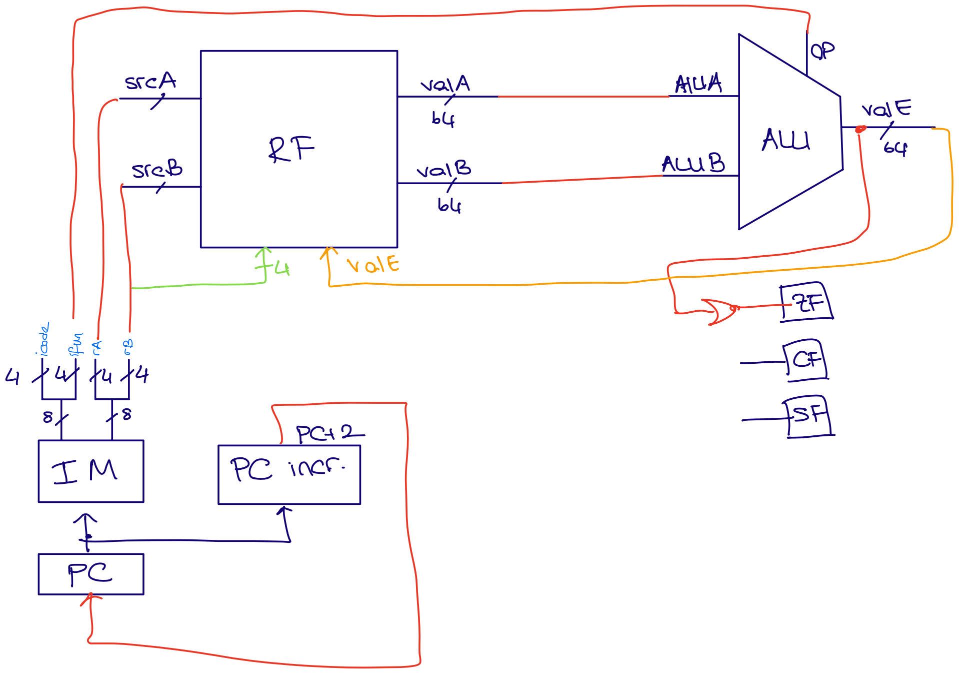




- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions

OPq rA, rB → Arithmetic Logic ops.





Stage Computation: Arith/Log. Ops

	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

rmmovq rA, D(rB)	4	0	rA	rB	D

1 byte 1 byte 8 byte

- **Fetch**
  - Read 10 bytes
- **Decode**
  - Read operand registers
- **Execute**
  - Compute effective address

D + rB

- **Memory**
  - Write to memory
- **Write back**
  - Do nothing
- **PC Update**
  - Increment PC by 10

Stage Computation: `rmmovq`

	`rmmovq` rA, D(rB)	

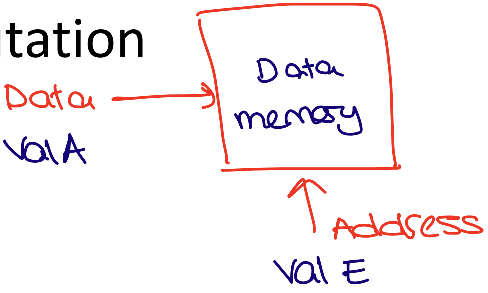
Fetch	icode:ifun ← M_1[PC]
	rA:rB ← M_1[PC+1]
	valC ← M_8[PC+2]
	valP ← PC+10 since 10 bytes are executed
		Read instruction byte
		Read register byte
		Read displacement D
		Compute next PC

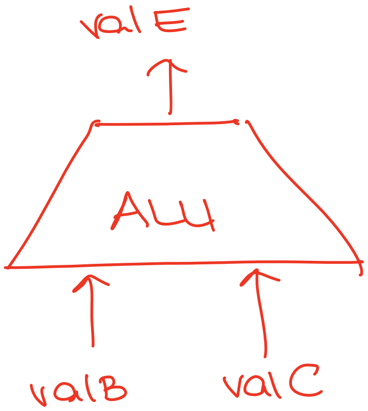
Decode	valA ← R[rA]
	valB ← R[rB]
		Read operand A
		Read operand B

Execute	valE ← valB + valC	Compute effective address
Memory	M_8[valE] ← valA	Write value to memory
Write back		
PC update	PC ← valP	Update PC

data memory

- Use ALU for address computation



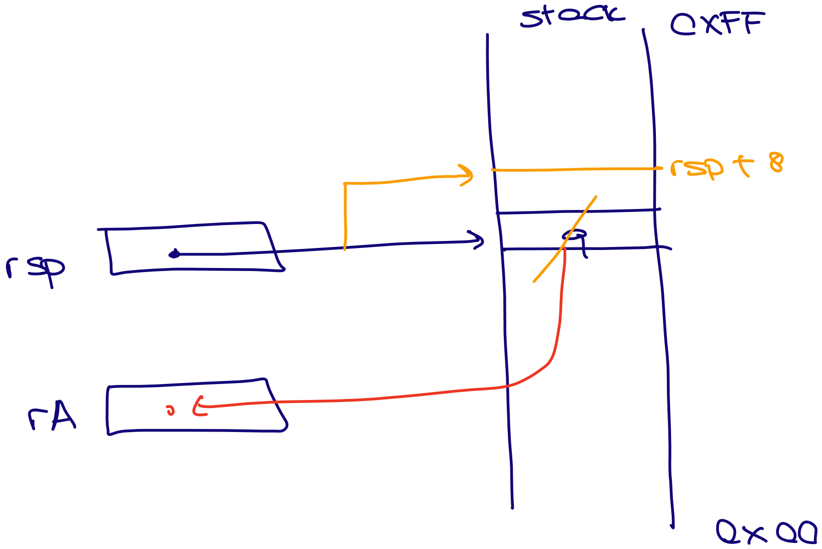


Executing popq

popq rA

b	0	rA	F

- **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] old rsp=valA
	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

`cmovXX rA, rB`

2	fn	rA	rB

- **Fetch**
  - Read 2 bytes
- **Decode**
  - Read operand registers
- **Execute**
  - If !cnd, then set destination register to 0xF

cnd is determined according to fn = function (== , >, <, ...)
then according to fn , Condition Codes (CC) determines cnd

- **Memory**
  - Do nothing
- **Write back**
  - Update register (or not)
- **PC Update**
  - Increment PC by 2

Stage Computation: Cond. Move

	cmovXX 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 ← 0
		Read operand A

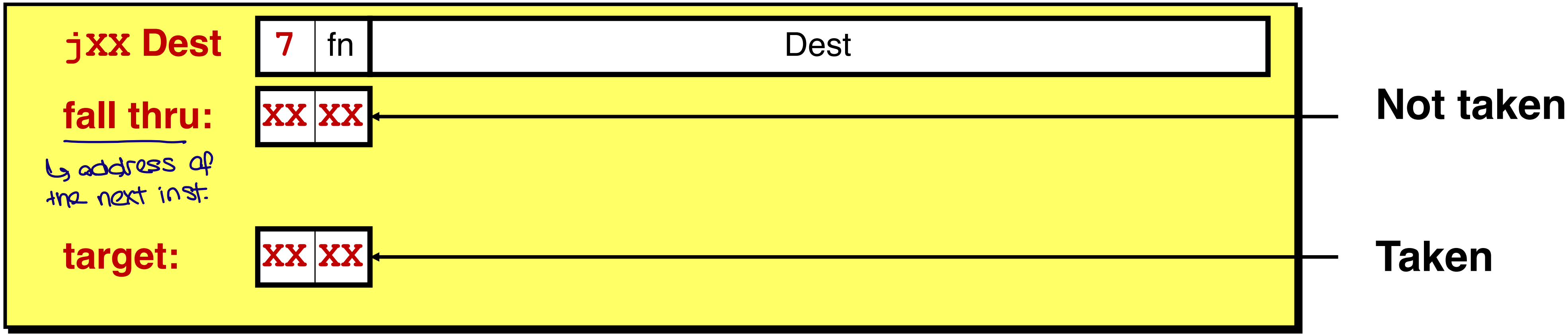
Execute	valE ← valB + valA If ! Cond(CC,ifun) rB ← 0xF	Pass valA through ALU (Disable register update)
Memory		
Write back	R[rB] ← valE	Write back result
PC update	PC ← valP	Update PC

→ rB =NULL if condition = false

will not be executed (disabled)

- 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

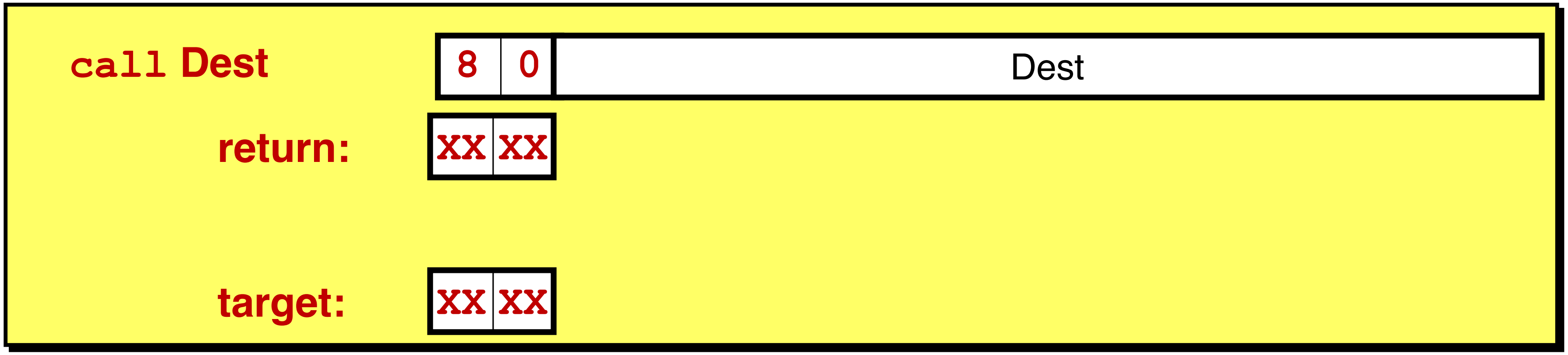
	jXX Dest	

Fetch	icode:ifun ← M_1[PC]
	valC ← M_8[PC+1]
	valP ← PC+9
		Read instruction byte
		Read destination address
		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`



- **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`

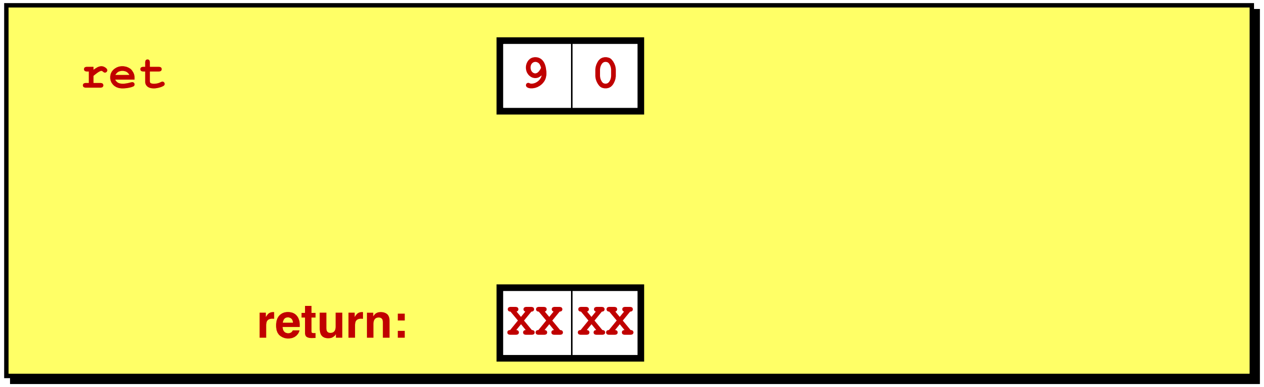
	`call` Dest	

Fetch	icode:ifun ← M_1[PC]
	valC ← M_8[PC+1]
	valP ← PC+9
		Read instruction byte
		Read destination address
		Compute return point

Decode	valB ← R[`%rsp`]	Read stack pointer
Execute	valE ← valB + –8	Decrement stack pointer
Memory	M_8[valE] ← valP	Write return value on stack
Write back	R[`%rsp`] ← 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 8

- **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[`%rsp`]
	valB ← R[`%rsp`]
		Read operand stack pointer
		Read operand stack pointer

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

		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

Computation Steps

		`call` Dest	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB		[Read register byte]
	valC	valC ← M_8[PC+1]	Read constant word
	valP	valP ← PC+9	Compute next PC
Decode	valA, srcA		[Read operand A]
	valB, srcB	valB ← R[`%rsp`]	Read operand B
Execute	valE	valE ← valB + –8	Perform ALU operation
	Cond code		[Set /use cond. code reg]
Memory	valM	M_8[valE] ← valP	Memory read/write
Write back	dstE	R[`%rsp`] ← 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

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

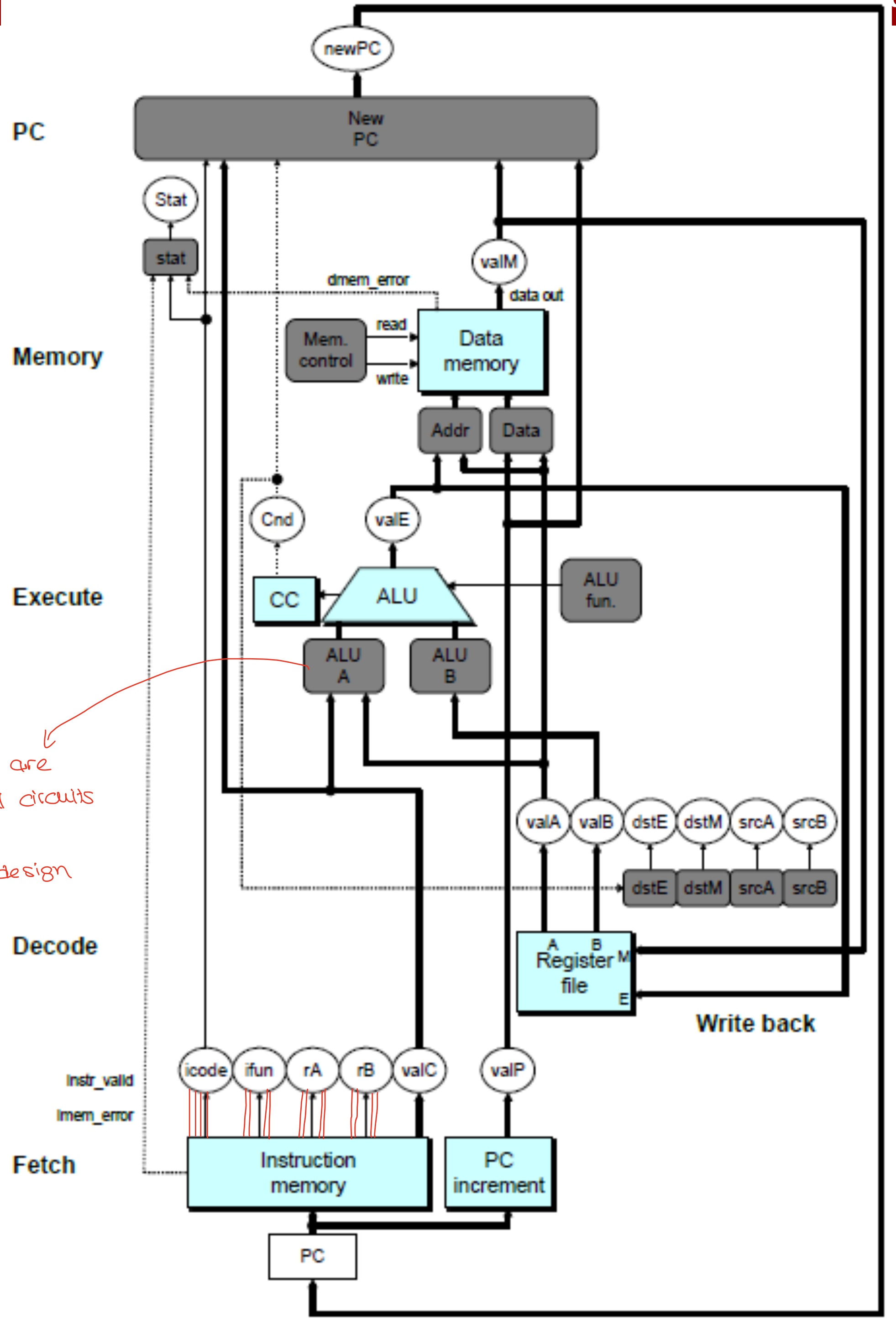
Memory elements → Register
PC
CC
Data Mem

We can read data from here anytime we want

Writing can be done only in the rising edge

SEQ Hardware

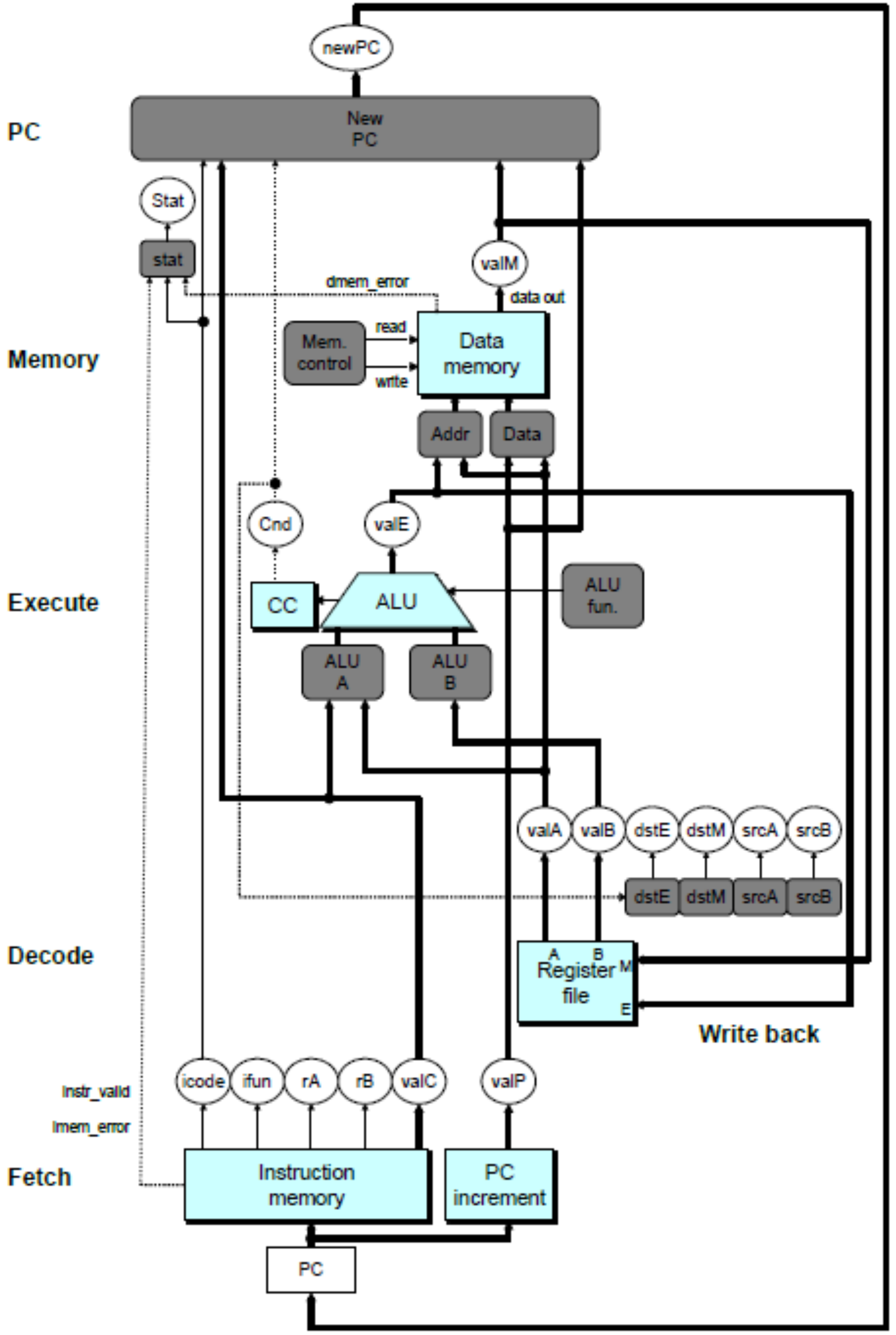
- **Key**
  - Blue boxes: predesigned hardware blocks
    - E.g., memories, ALU



gray boxes are combinational circuits
↓
we will design them

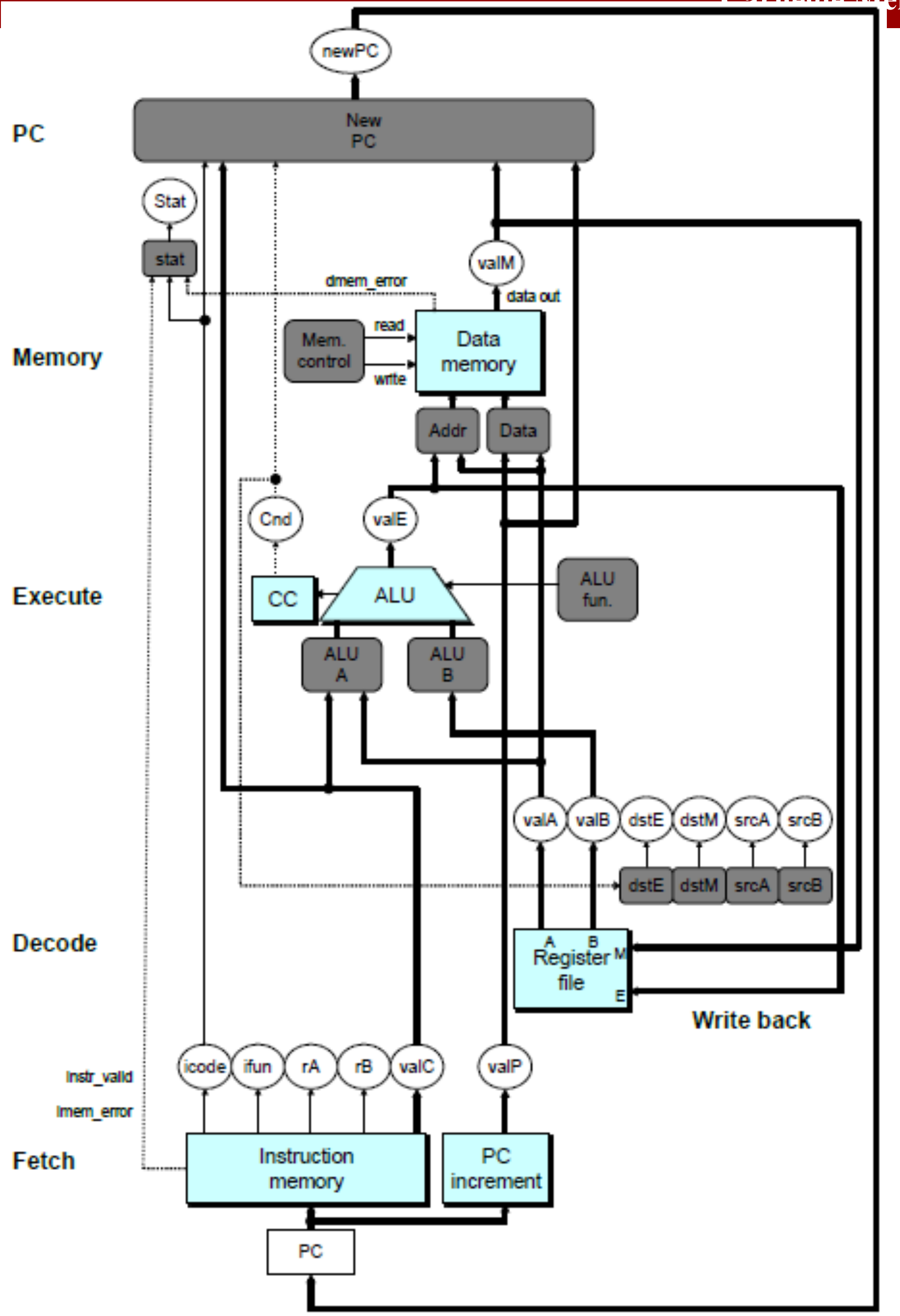
SEQ Hardware

- **Key**
  - White ovals: labels for signals
  - Thick lines: 64-bit word values
  - Thin lines: 4-8 bit values
  - Dotted lines: 1-bit values



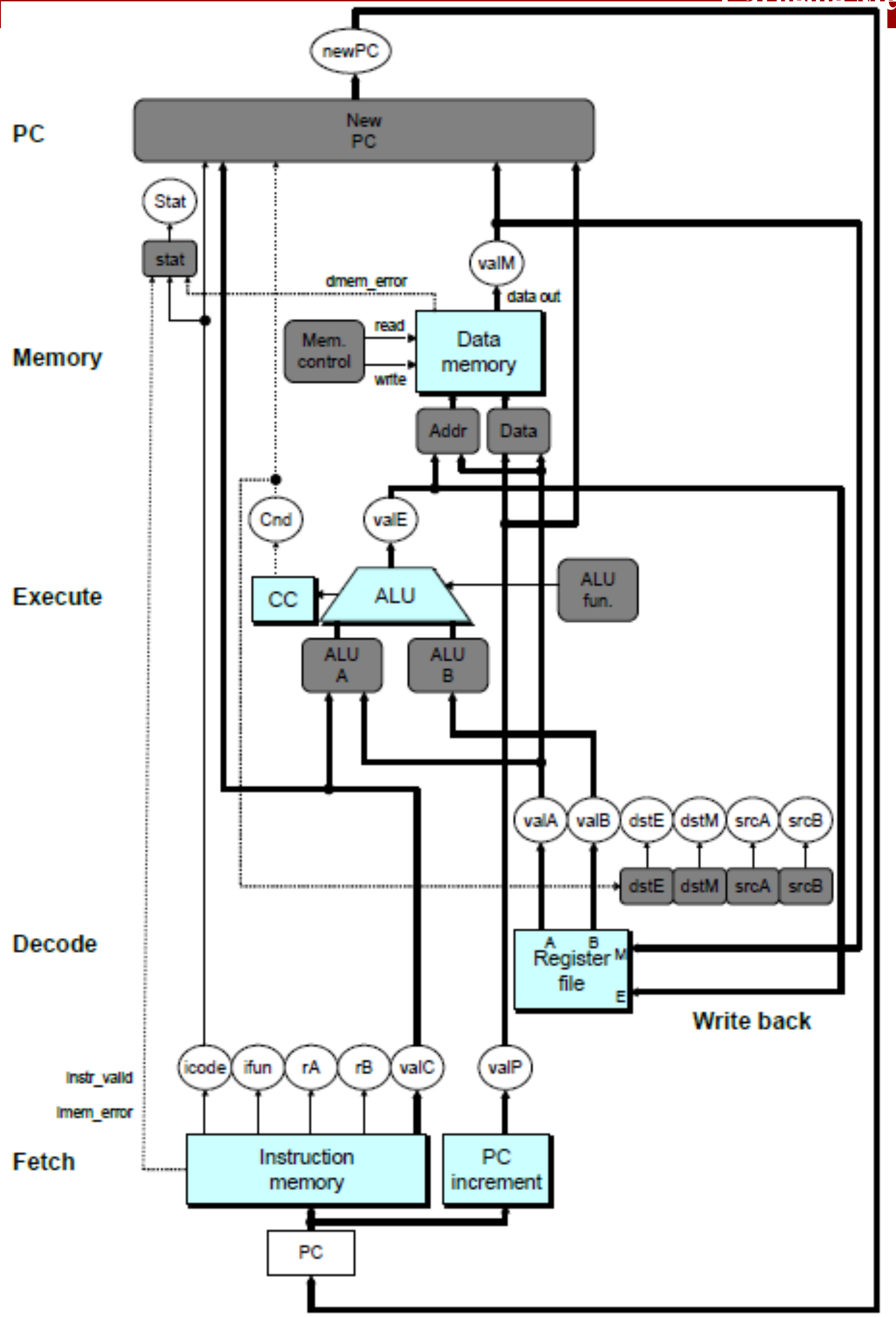
SEQ Hardware

- **Key**
  - Gray boxes: control logic
    - Describe in HCL

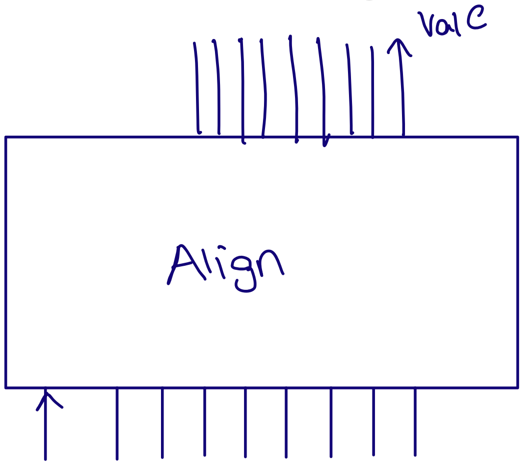


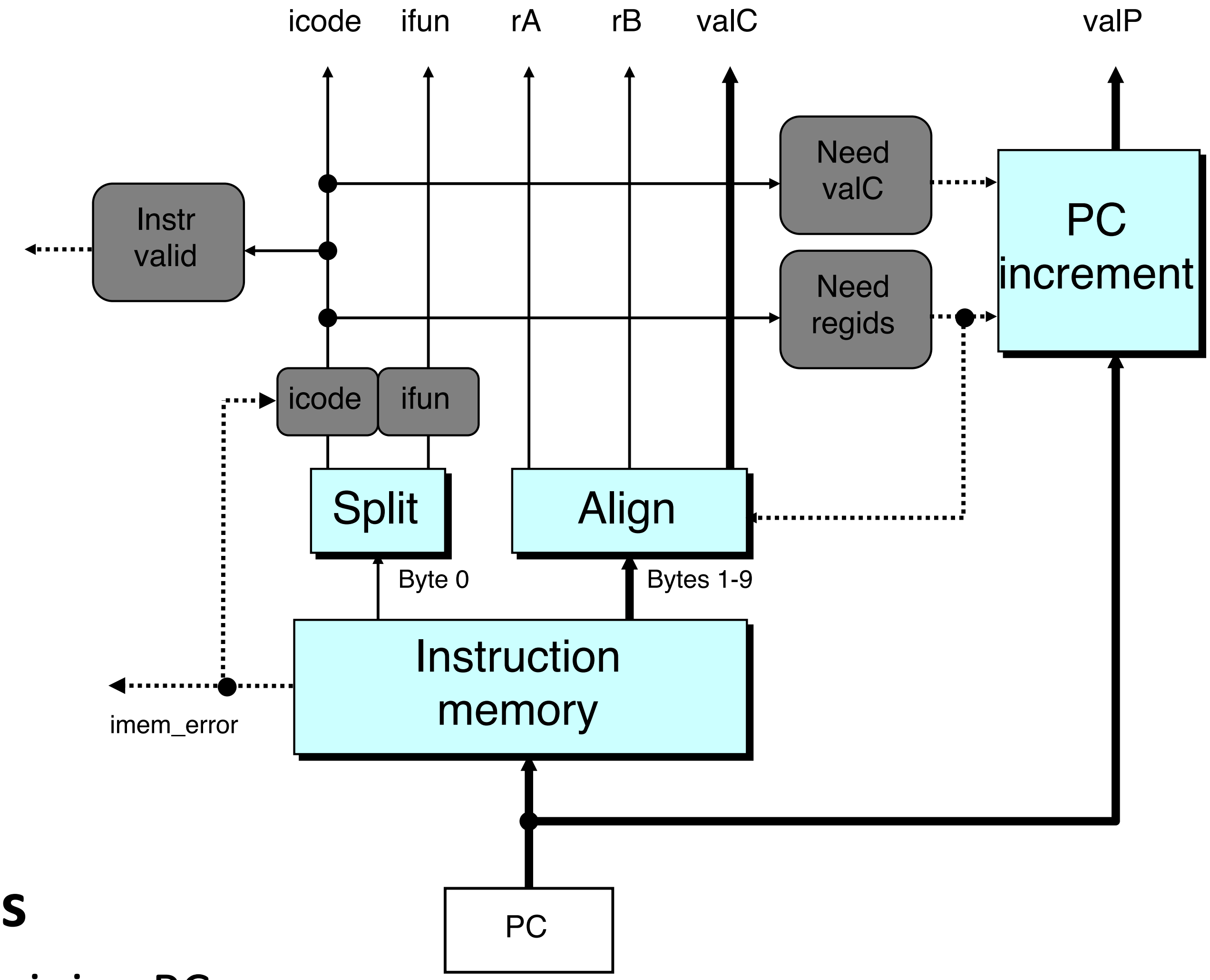
SEQ Hardware

- **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



Fetch Logic





Predefined Blocks

- PC: Register containing PC
- Instruction memory: Read 10 bytes (PC to PC+9)
  - Signal invalid address

ALU

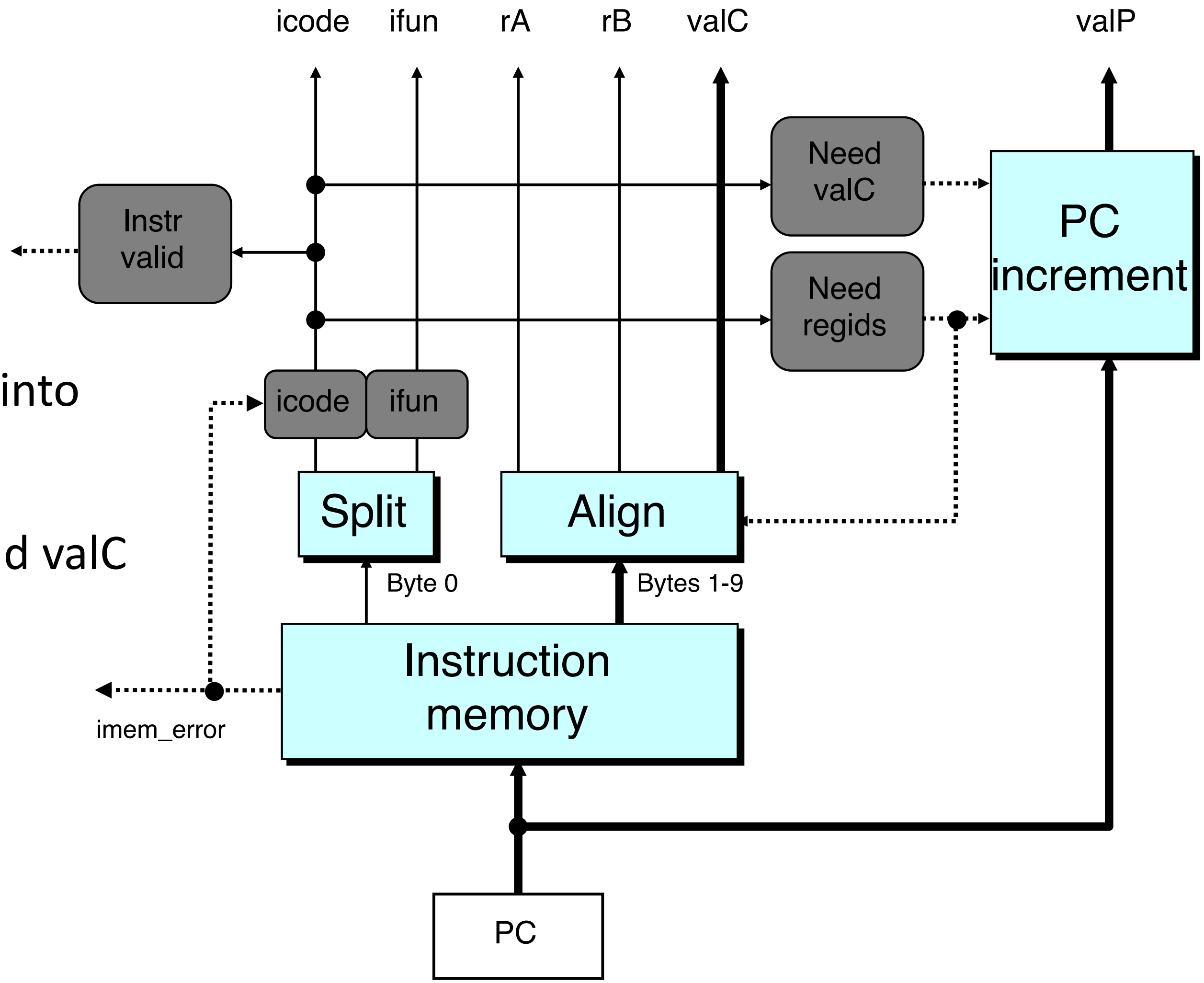
ic	if	rA	rB

immovq rA D(rB)

Align does this →

ic	if	rA	rB	valC

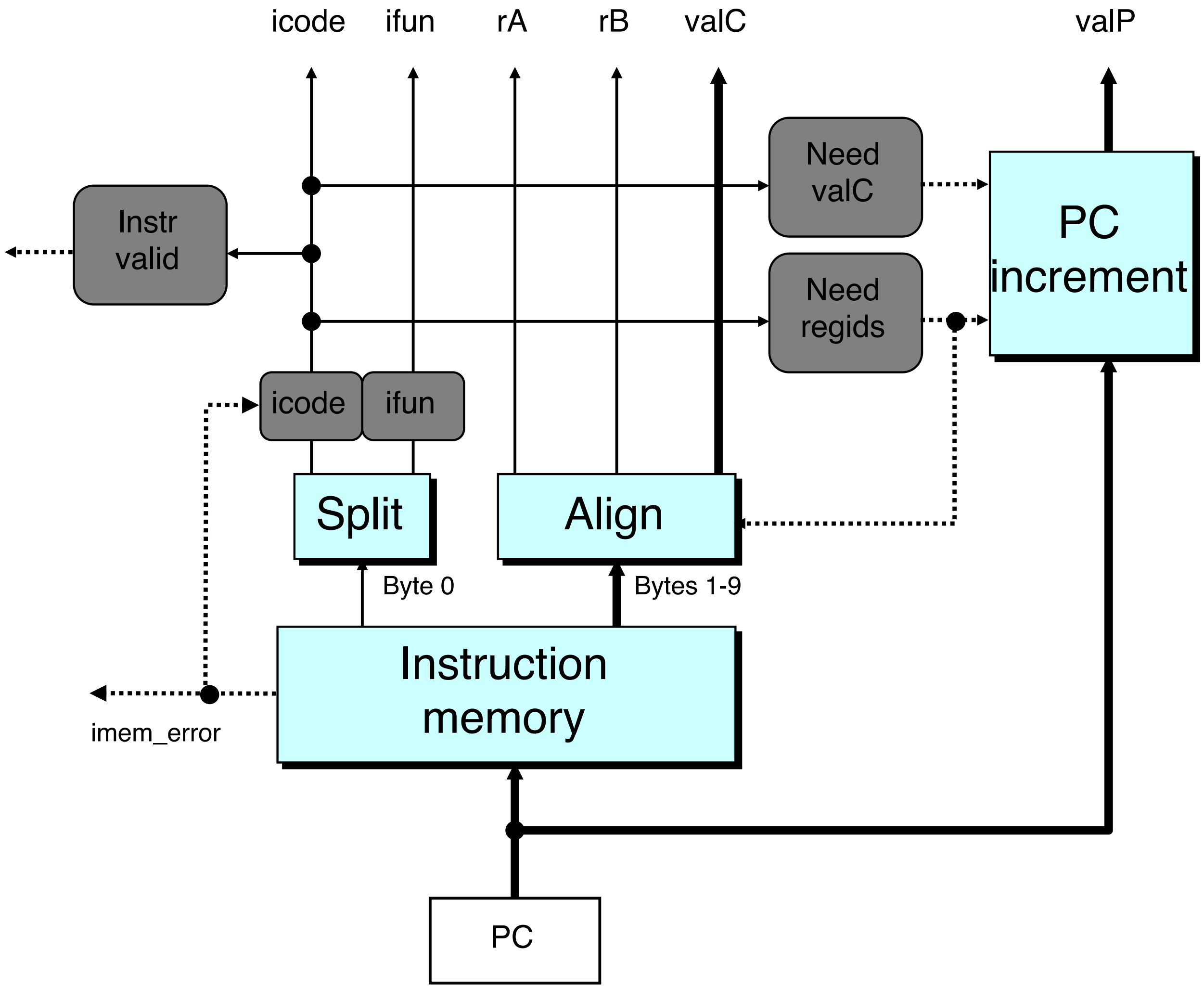
Fetch Logic



Split: Divide instruction byte into icode and ifun

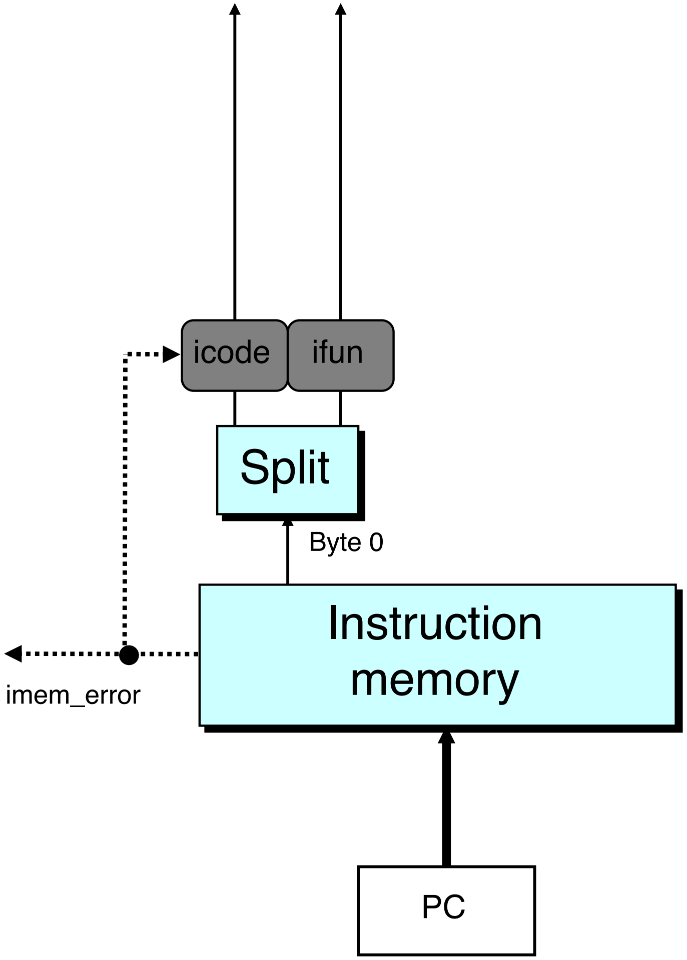
Align: Get fields for rA, rB, and valC

Fetch Logic



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

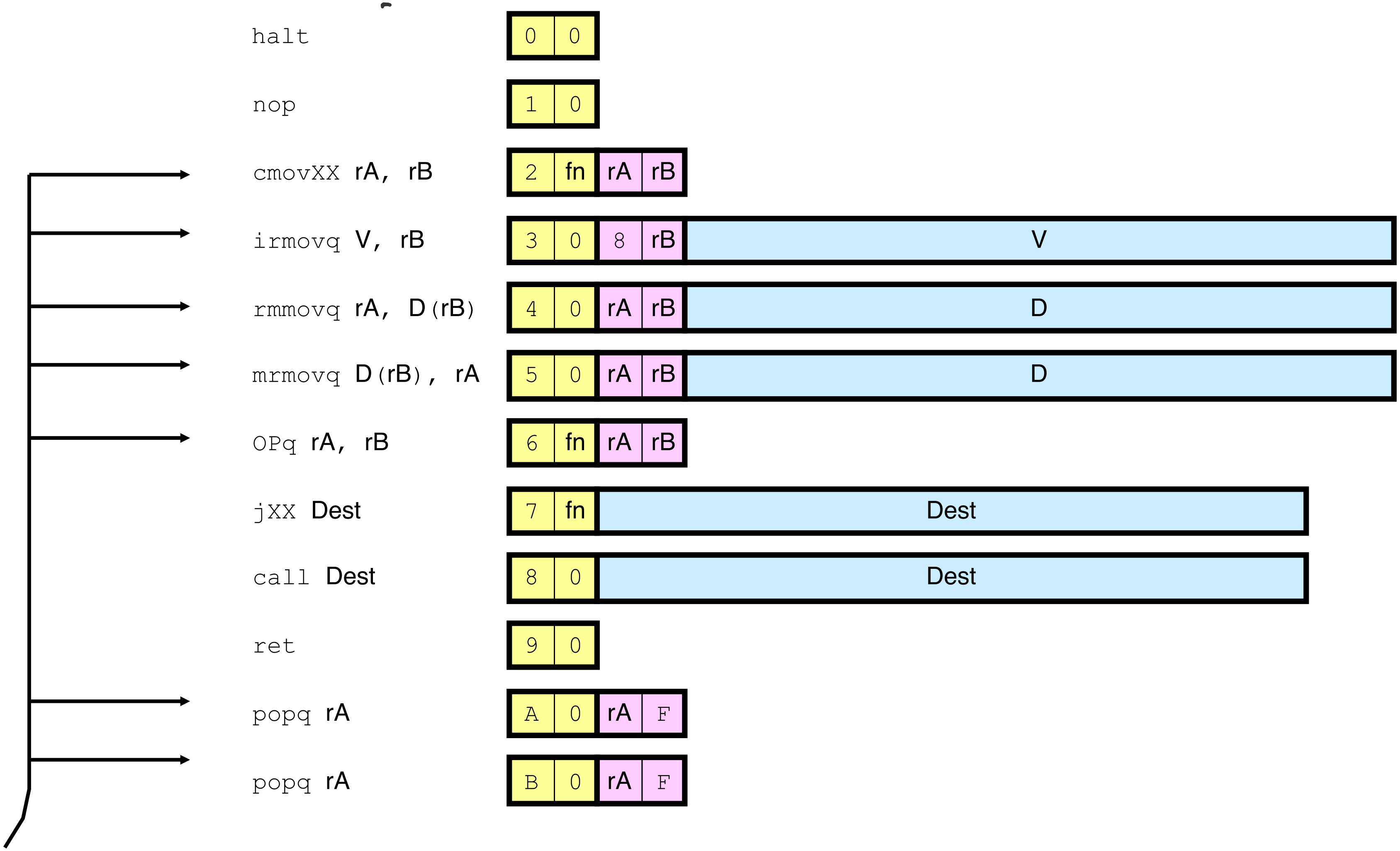
Fetch Control Logic in HCL



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

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

Fetch Control Logic in HCL



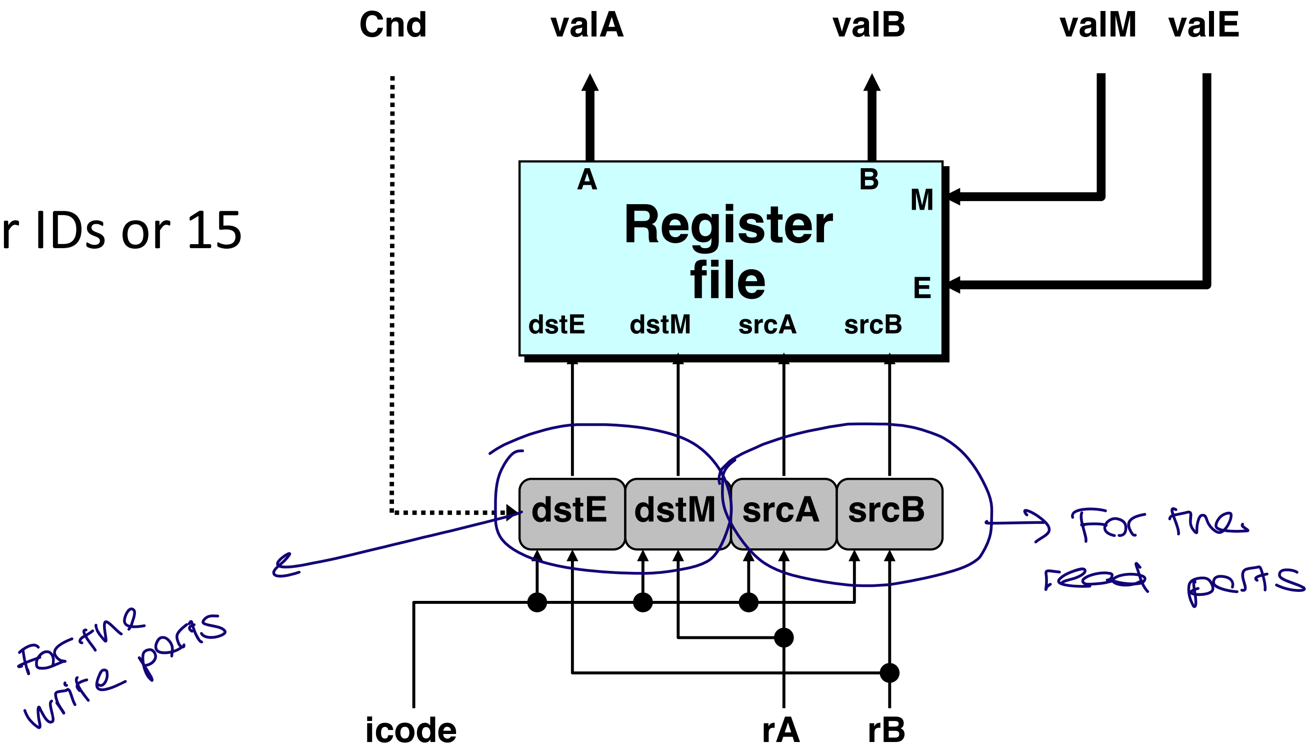
```
bool need_regids =
        icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
                   IIRMOVQ, IRMMOVQ, IMRMOVQ };

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

→ Toplam 1→9 [illegible] [illegible] → icode [illegible]'dan farklıysa invalid

Decode Logic

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



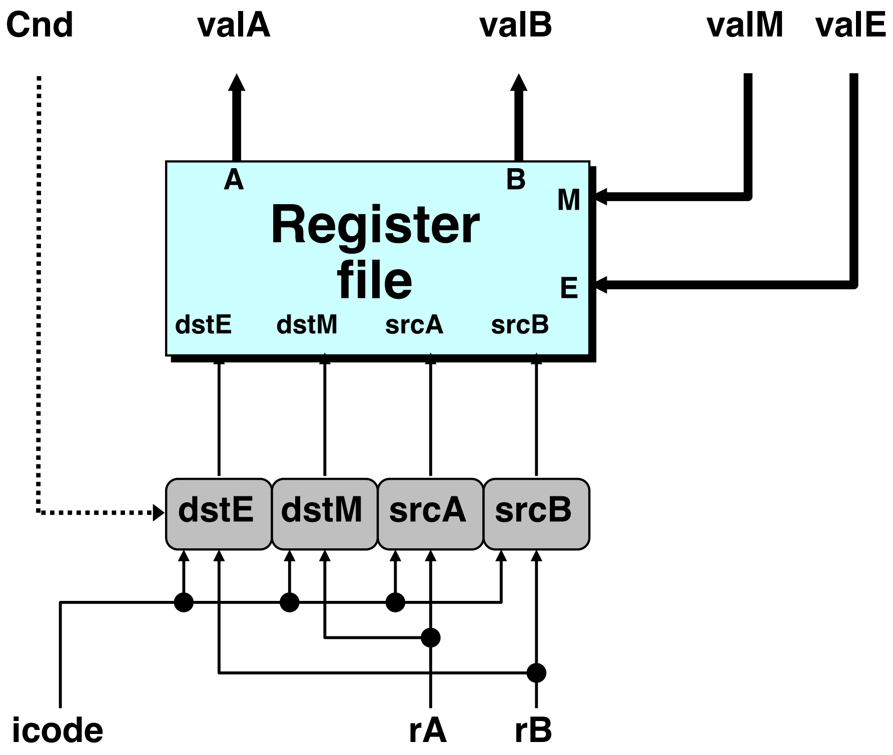
For the write ports

For the read ports

Decode Logic

Control Logic

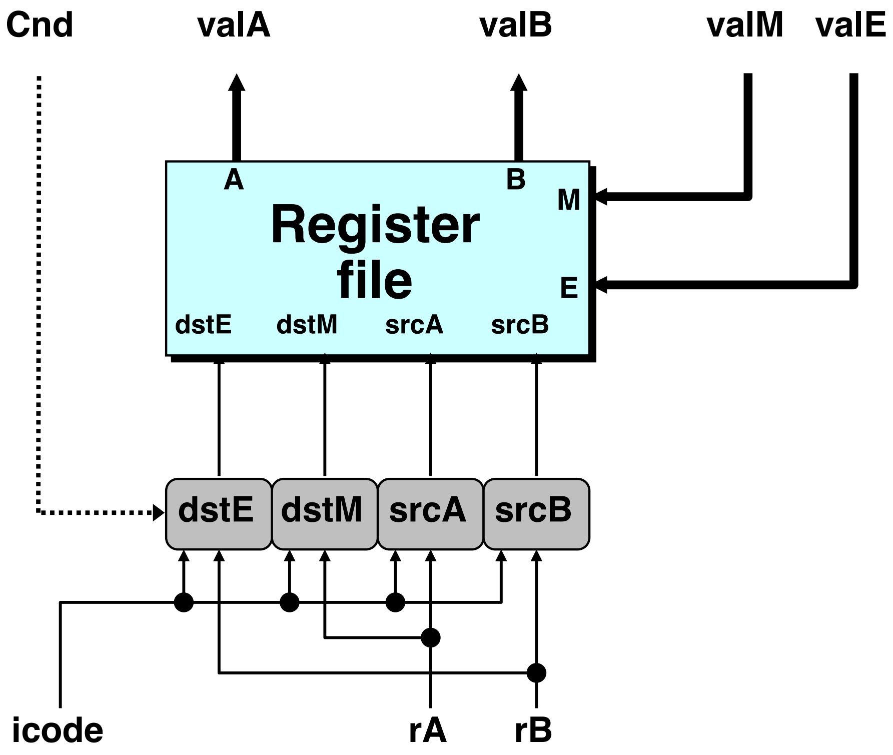
- srcA, srcB: read port addresses
- dstE, dstM: write port addresses



Decode Logic

Signals

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



Decode Logic

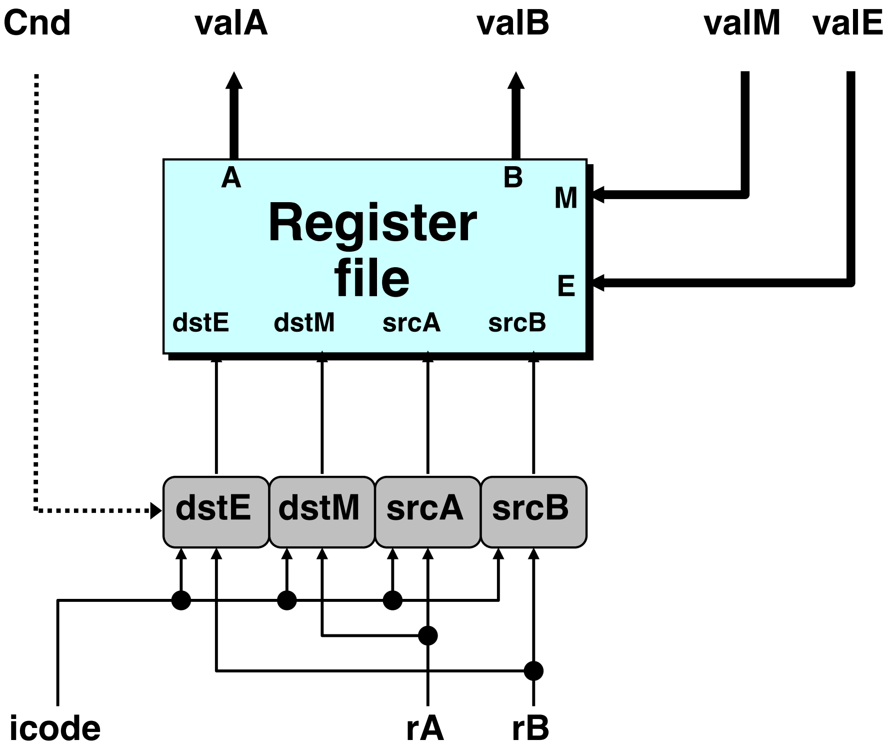
- **Register File**
  - Read ports A, B
  - Write ports E, M
  - Addresses are register IDs or 15 (0xF) (no access)

Control Logic

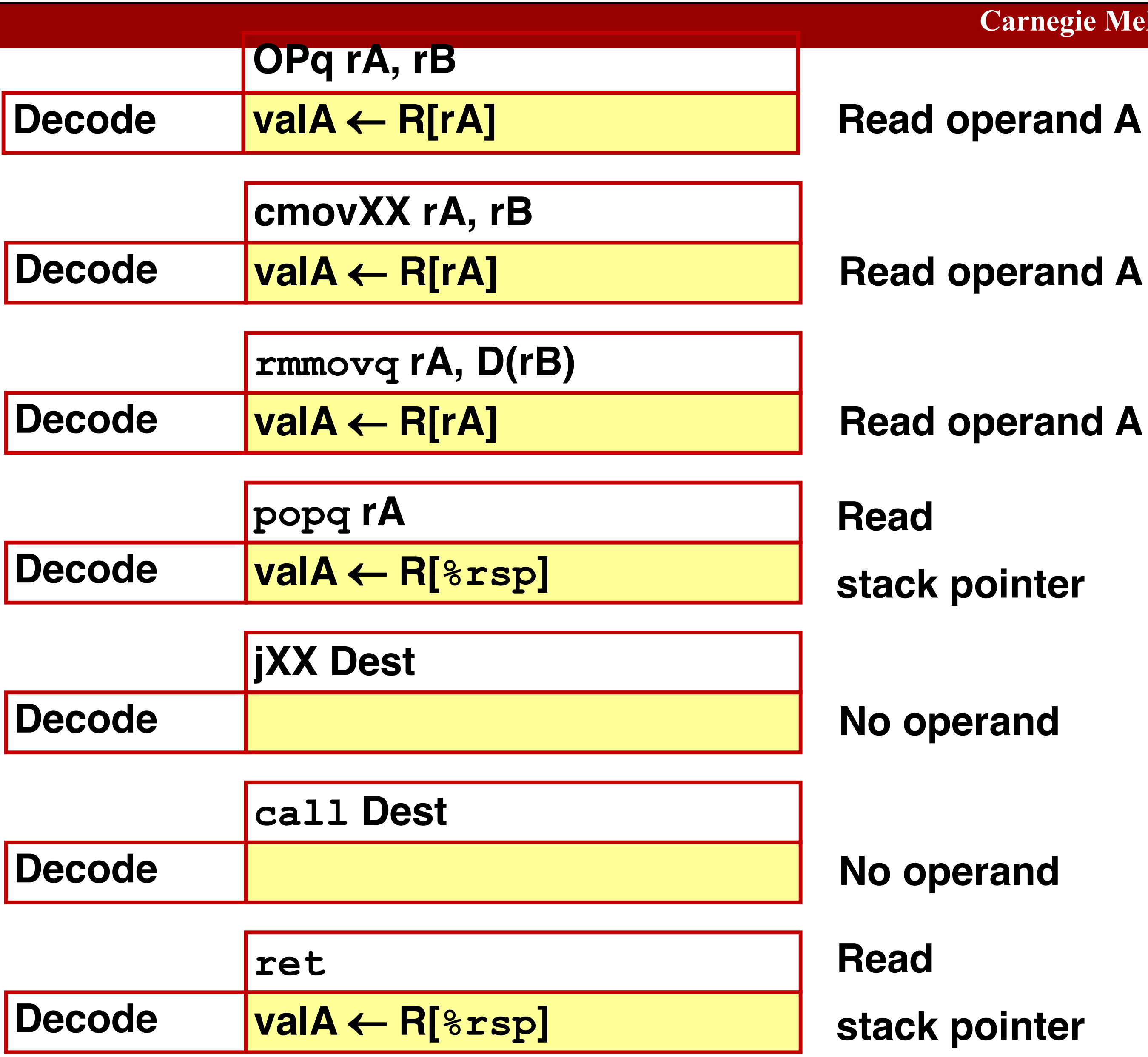
- **srcA, srcB:** read port addresses
- **dstE, dstM:** write port addresses

Signals

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



A Source



OPq rA, rB

Decode | valA ← 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
];
```

E Destination

update register files

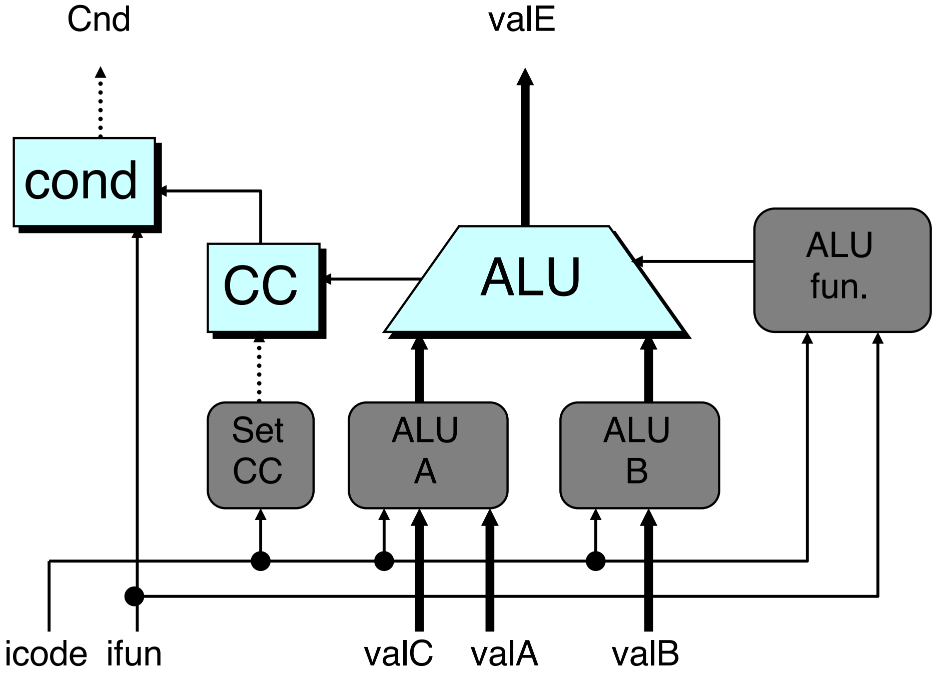
	OPq rA, rB	
Write-back	R[rB] ← valE	Write back result
	cmovXX rA, rB	Conditionally
Write-back	R[rB] ← valE	write back result
	rmmovq rA, D(rB)	
Write-back		None
	popq rA	Update
Write-back	R[%rsp] ← valE	stack pointer
	jXX Dest	
Write-back		None
	call Dest	Update
Write-back	R[%rsp] ← valE	stack pointer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

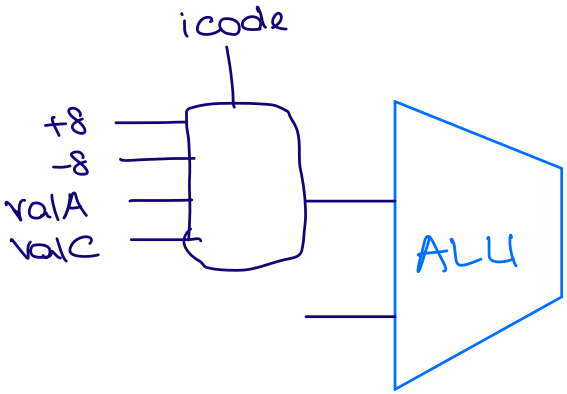
Execute Logic

condition codes will be updated only by ALU

- **Units**
  - ALU
    - Implements 4 required functions
    - Generates condition code values
  - CC
    - Register with 3 condition code bits
  - cond
    - Computes conditional jump/move flag
- **Control Logic**
  - Set CC: Should condition code register be loaded?
  - ALU A: Input A to ALU
  - ALU B: Input B to ALU
  - ALU fun: What function should ALU compute?



ALU A Input



	OPq rA, rB	
Execute	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

→ immediate to mem

```
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

	OPI 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

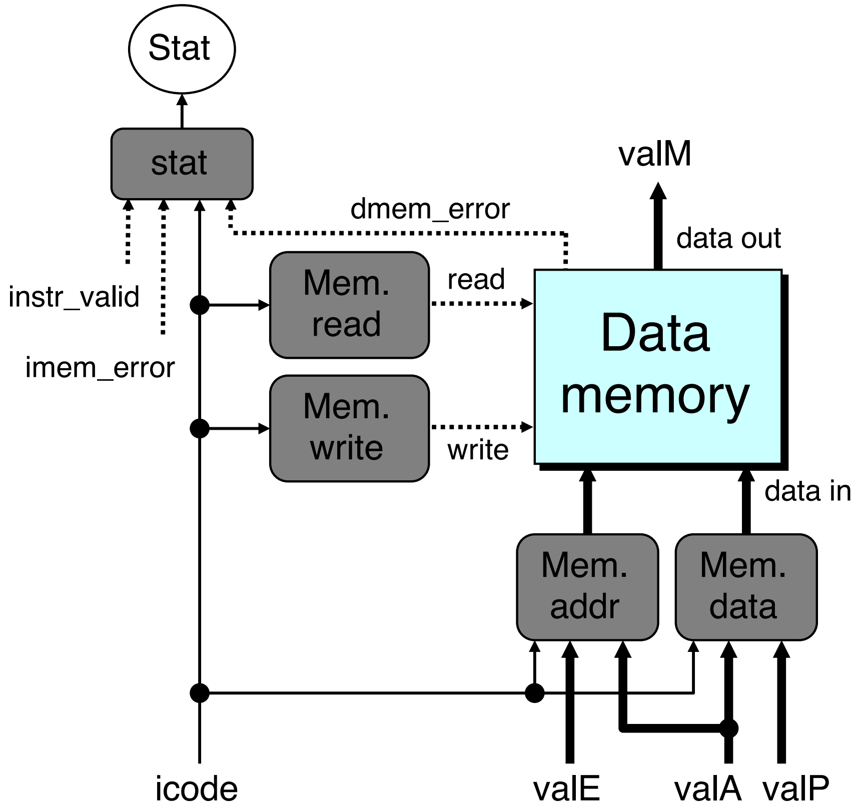
	ret
Execute	valE ← valB + 8

Increment stack pointer

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

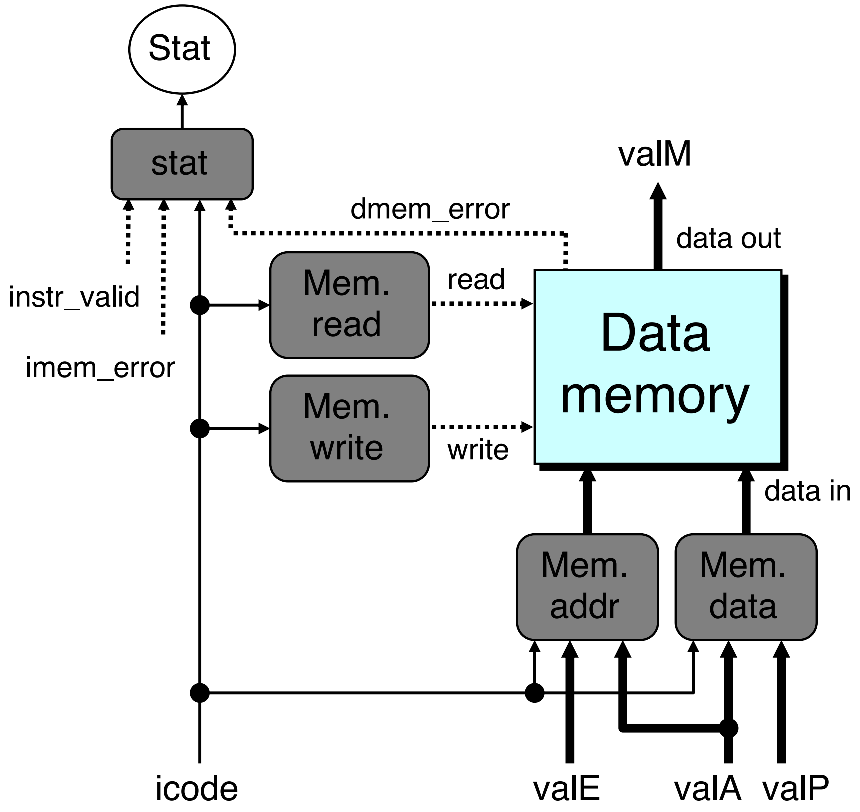
Memory Logic

- **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



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;
];
```

status halt

status all

ok

Memory Address

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

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

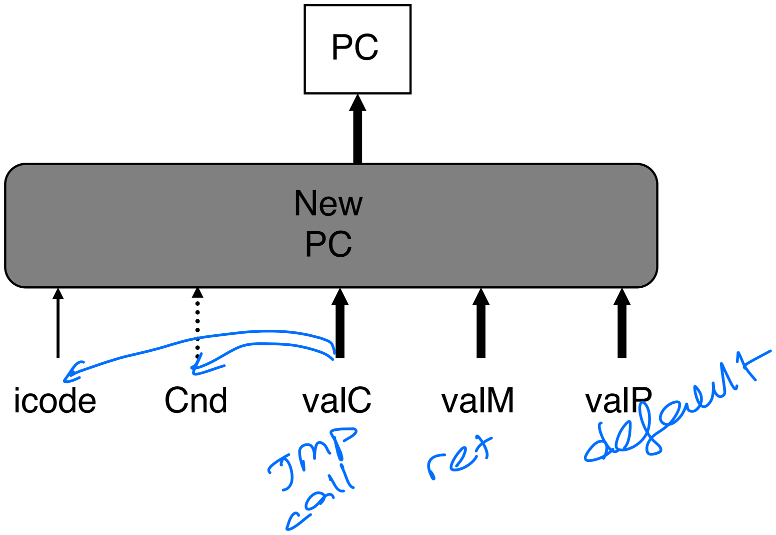
Memory Read

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

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

PC Update Logic

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

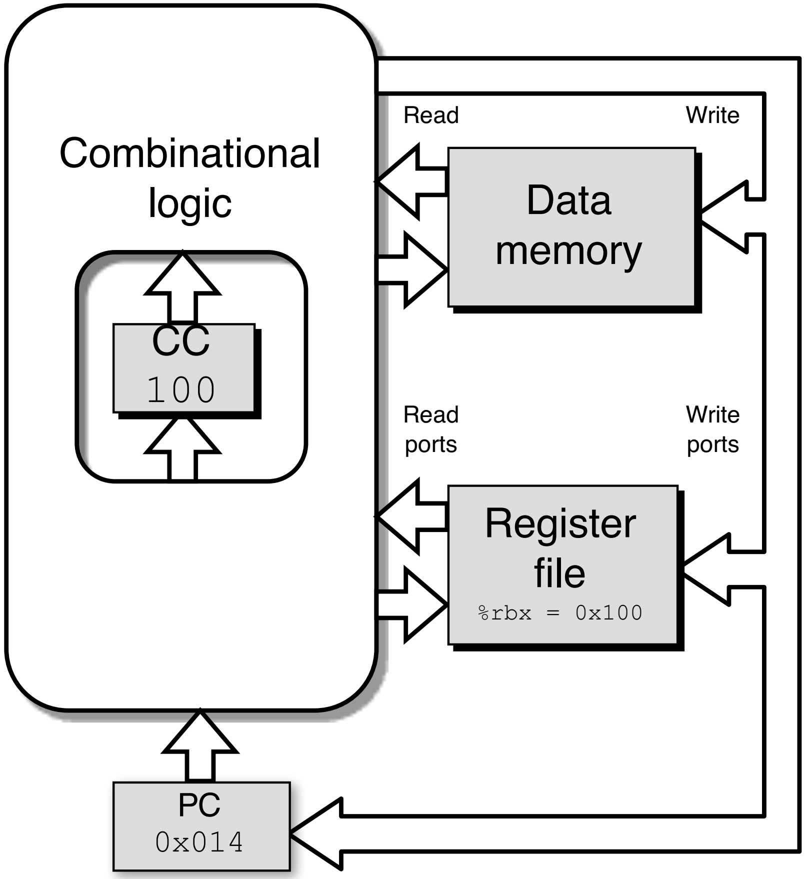


PC Update

	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;
];
```

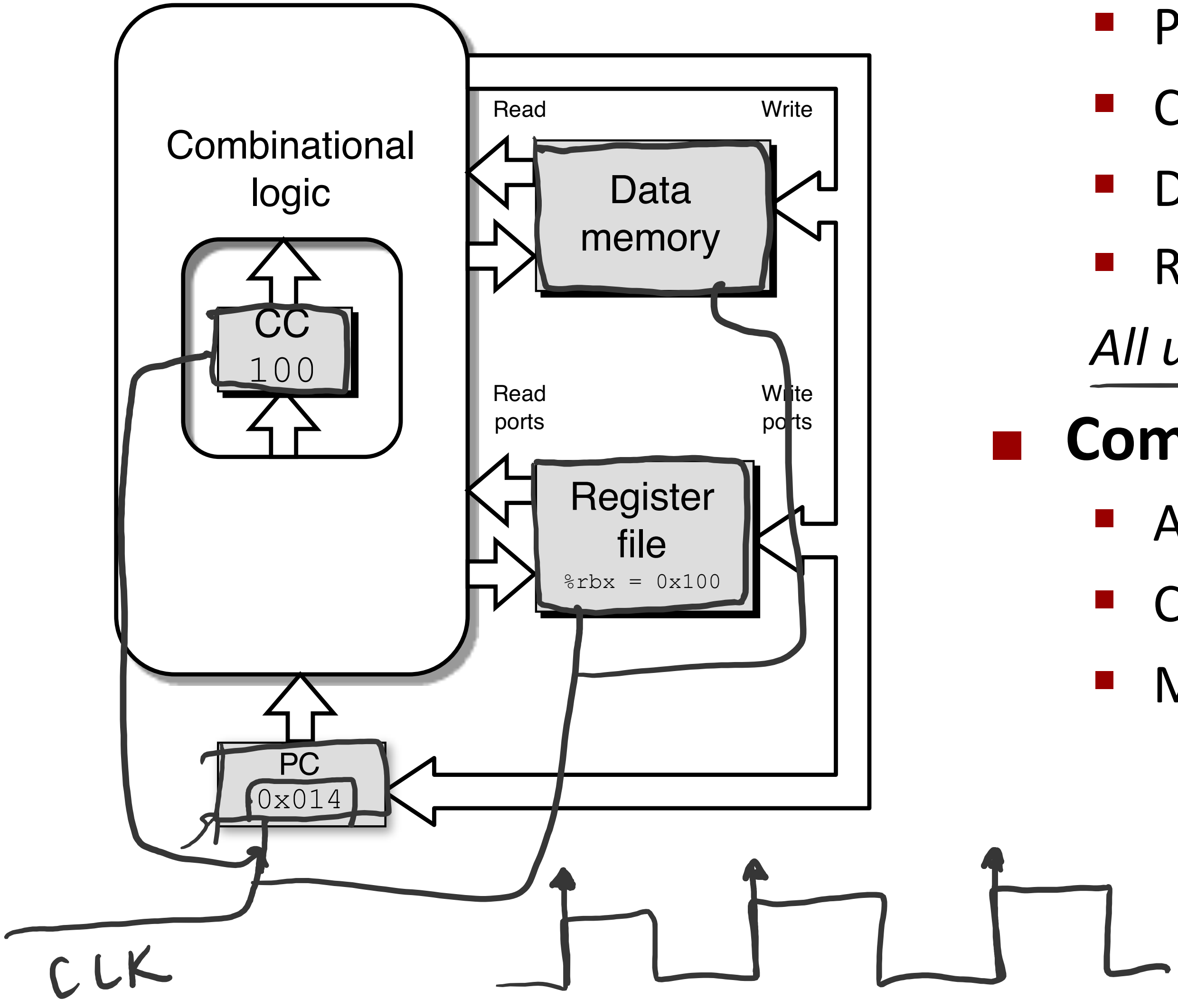
SEQ Operation



- **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

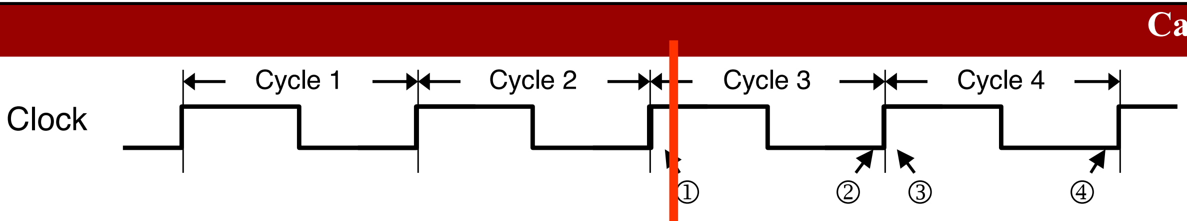


- **State**
  - PC register
  - Cond. Code register
  - Data memory
  - Register file

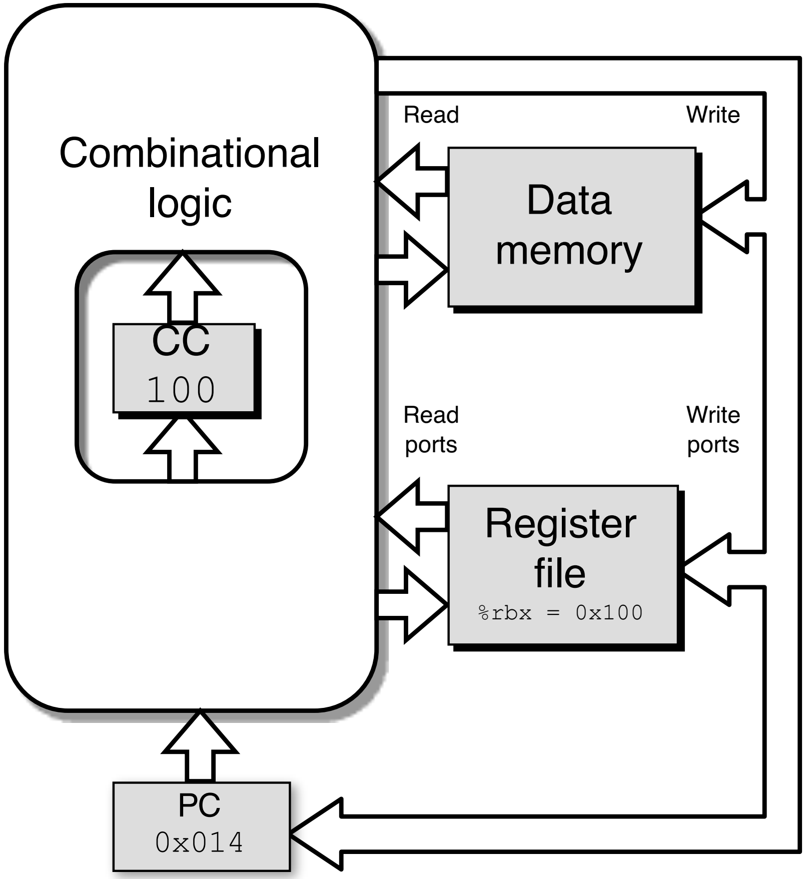
  All updated as clock rises

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

SEQ Operation #2

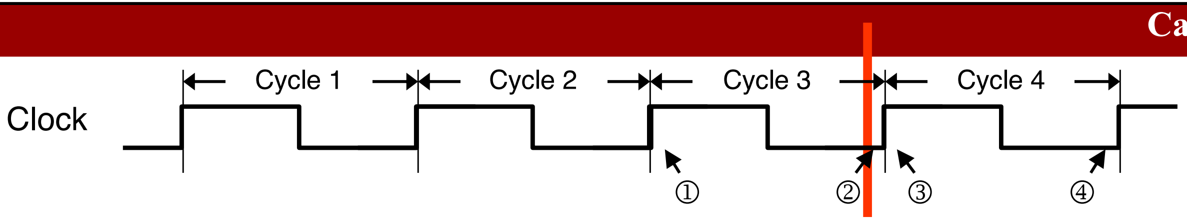


Cycle 1:	`0x000:`	`irmovq $0x100,%rbx`	`# %rbx <-- 0x100`
Cycle 2:	`0x00a:`	`irmovq $0x200,%rdx`	`# %rdx <-- 0x200`
Cycle 3:	`0x014:`	`addq %rdx,%rbx`	`# %rbx <-- 0x300 CC <-- 000`
Cycle 4:	`0x016:`	`je dest`	`# Not taken`
Cycle 5:	`0x01f:`	`rmmovq %rbx,0(%rdx)`	`# M[0x200] <-- 0x300`

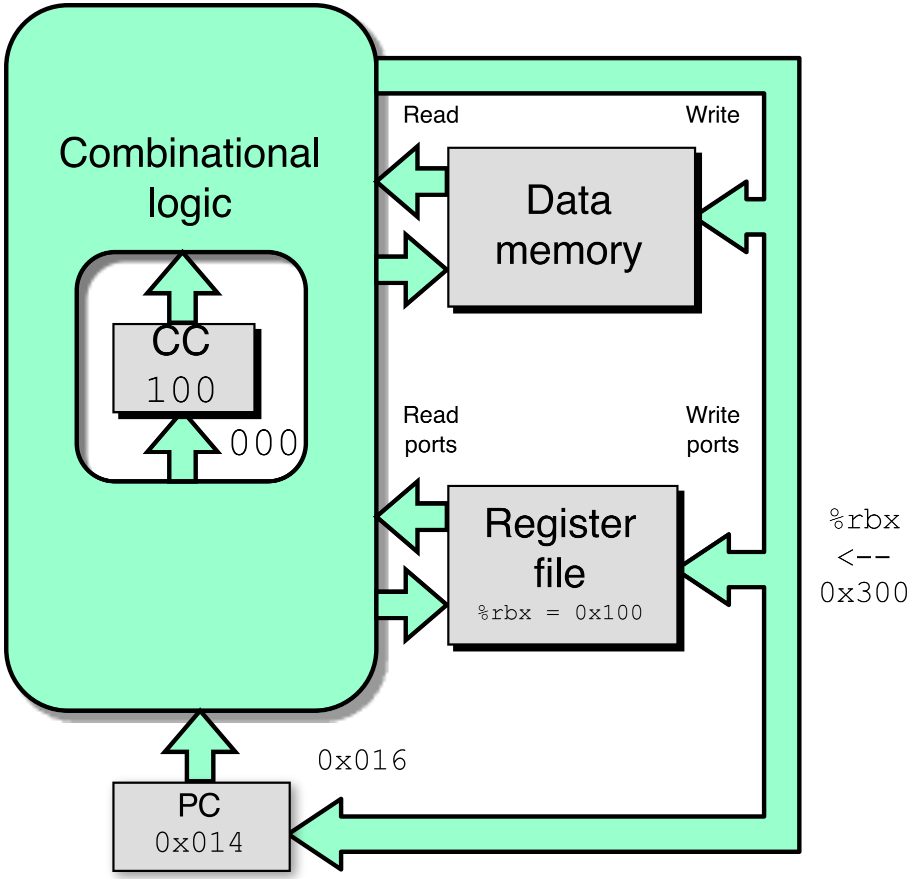


- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes

SEQ Operation #3

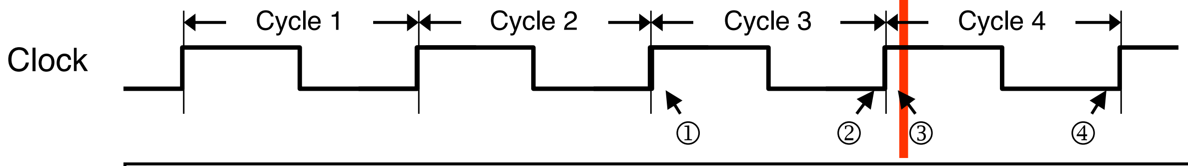


Cycle 1:	`0x000:    irmovq $0x100,%rbx   # %rbx <-- 0x100`
Cycle 2:	`0x00a:    irmovq $0x200,%rdx   # %rdx <-- 0x200`
Cycle 3:	`0x014:    addq %rdx,%rbx       # %rbx <-- 0x300 CC <-- 000`
Cycle 4:	`0x016:    je dest              # Not taken`
Cycle 5:	`0x01f:    rmmovq %rbx,0(%rdx)  # M[0x200] <-- 0x300`

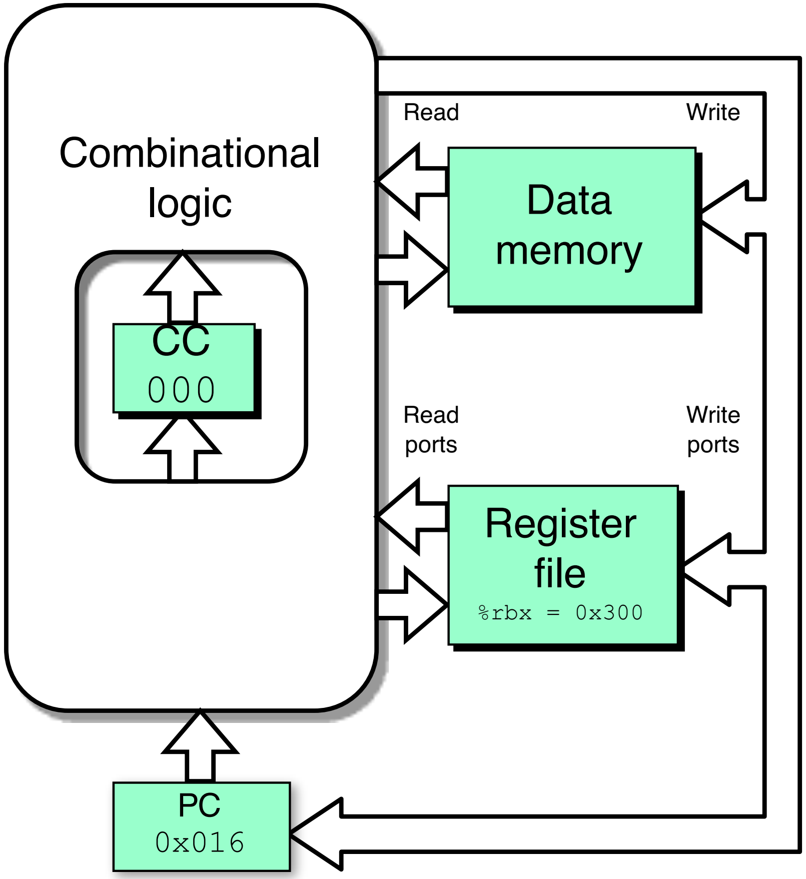


- state set according to second `irmovq` instruction
- combinational logic generates results for `addq` instruction

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

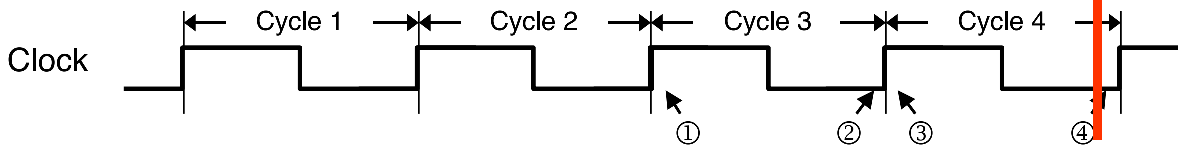


- state set according to `addq` instruction
- combinational logic starting to react to state changes

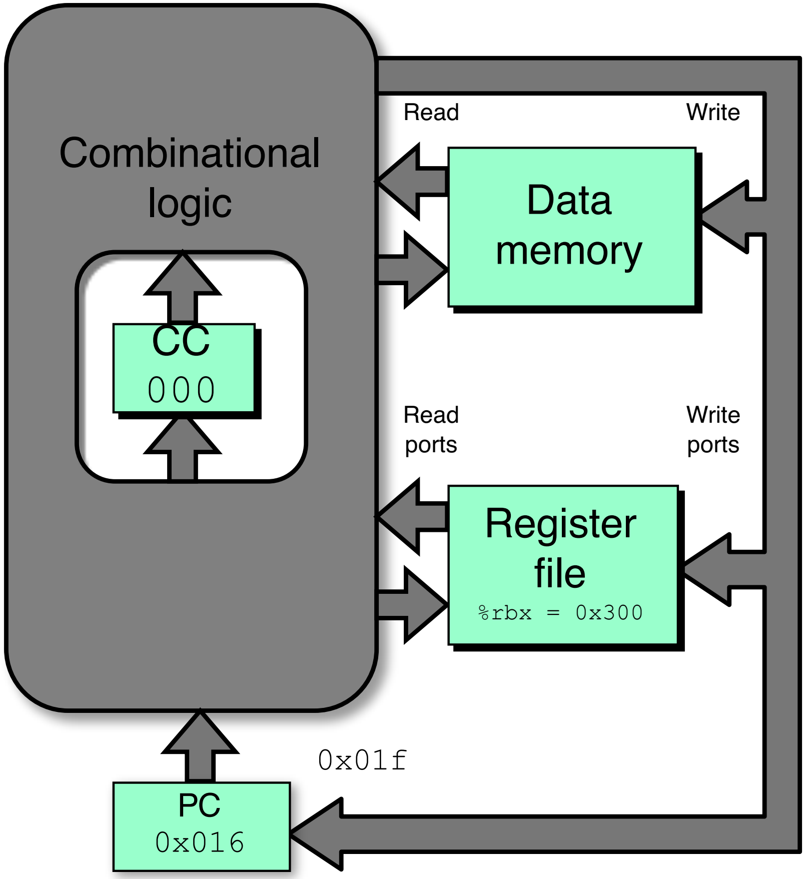
multiple instructions at one clock cycle

Midtermde HCL kodlarını tamamen anlamamıza gerek yok

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