

CS:APP Chapter 4
Computer Architecture
Sequential
Implementation

Randal E. Bryant

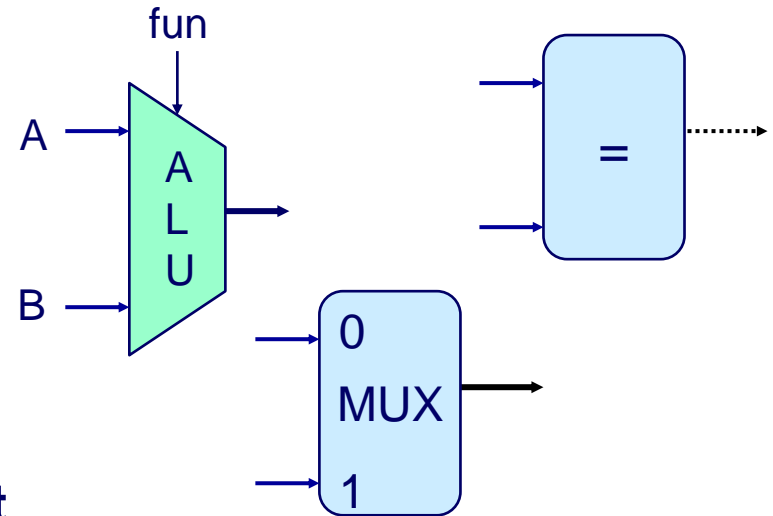
Carnegie Mellon University

<http://csapp.cs.cmu.edu>

Building Blocks

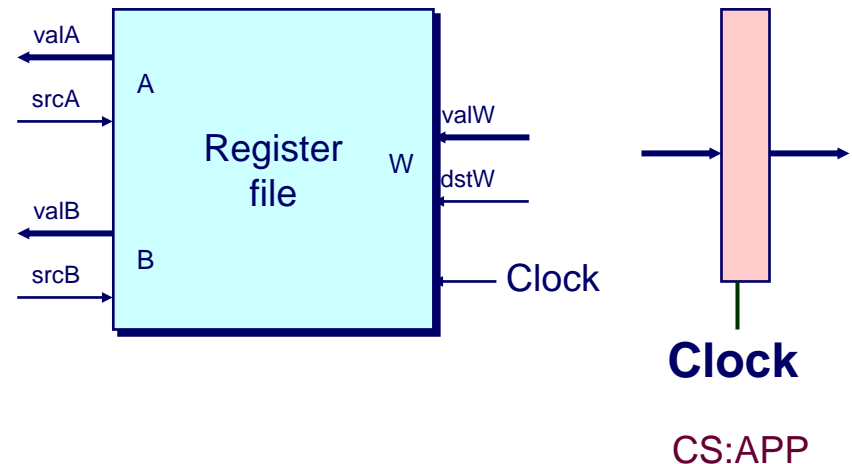
Combinational Logic

- 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



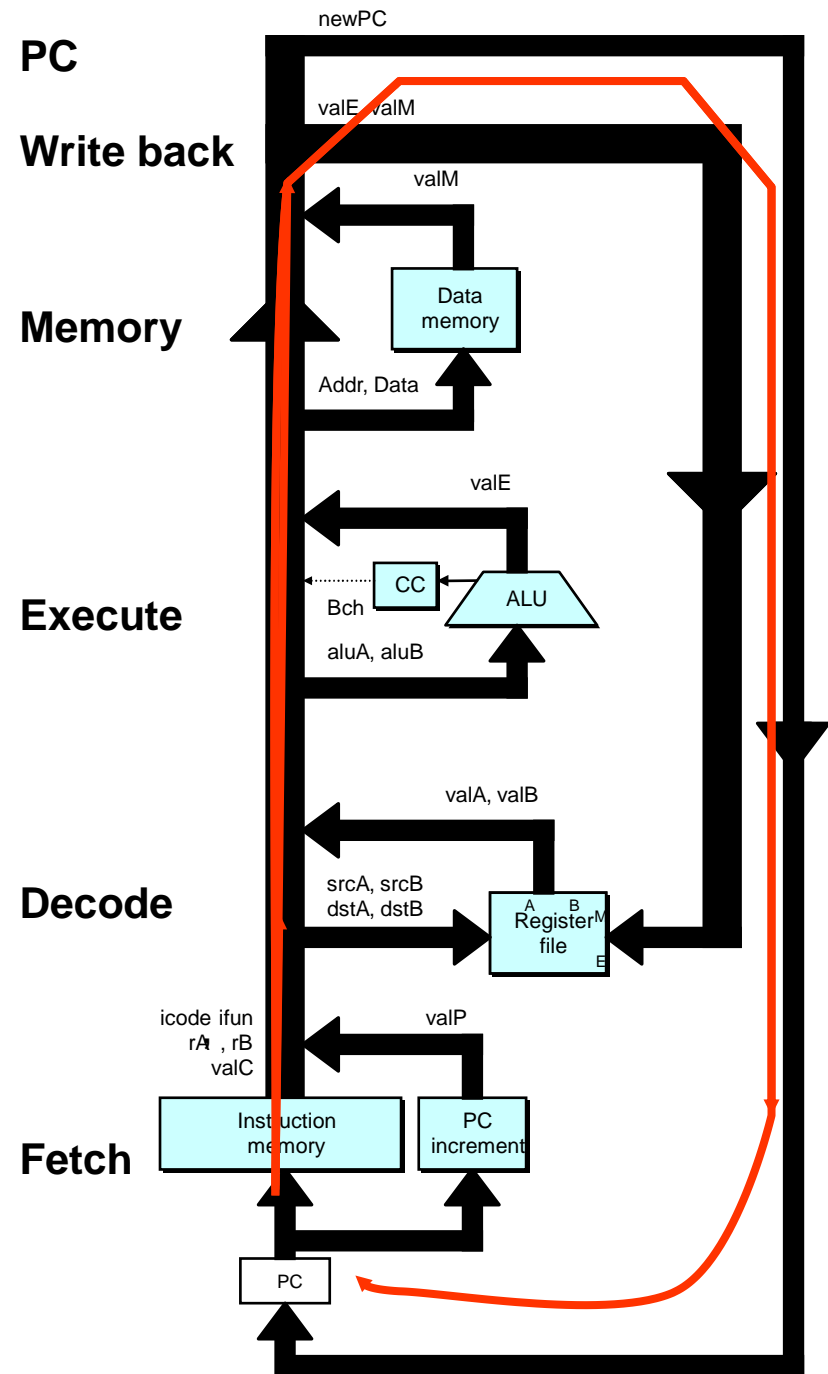
SEQ Hardware Structure

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 Stages

Fetch

- Read instruction from instruction memory

Decode

- Read program registers

Execute

- Compute value or address

Memory

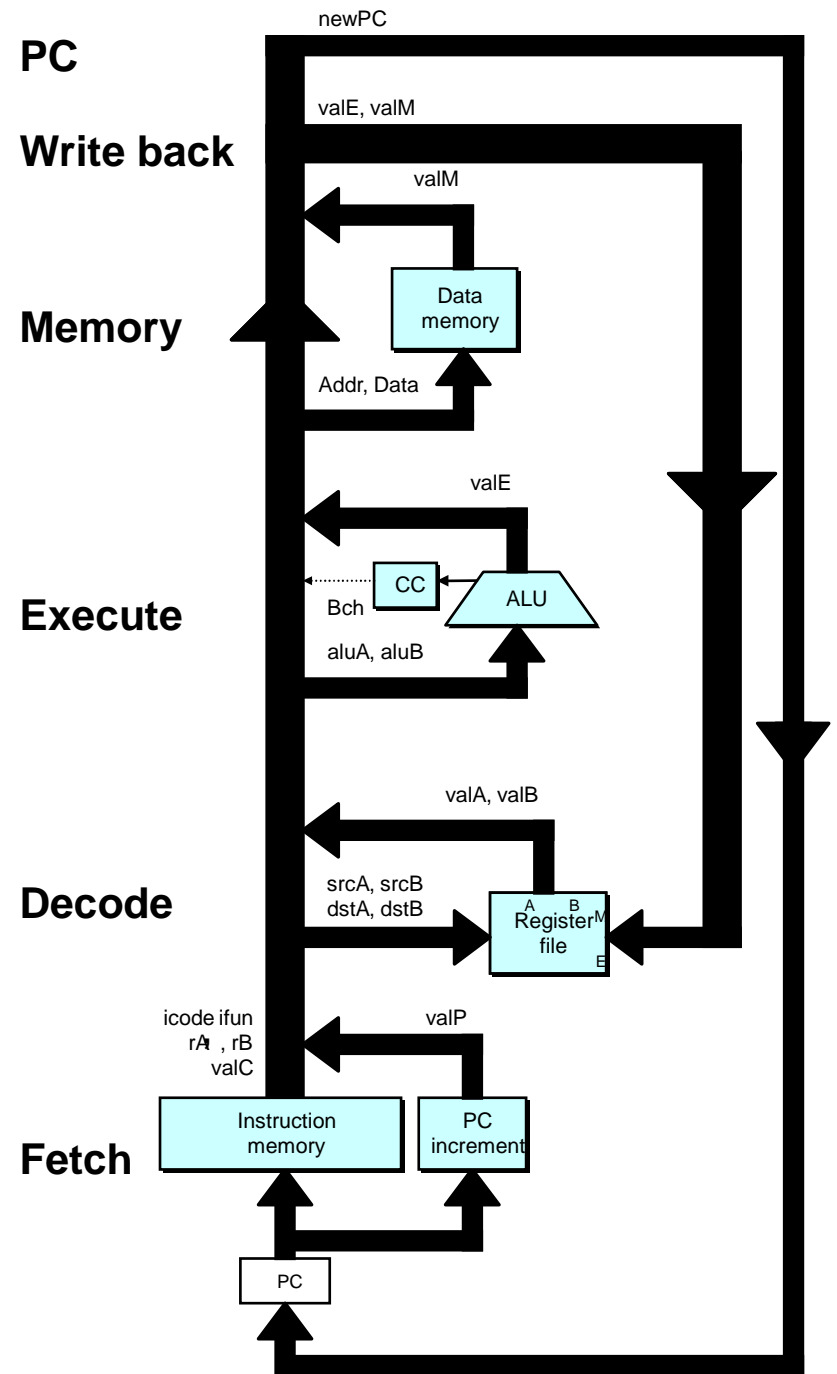
- Read or write data

Write Back

- Write program registers

PC

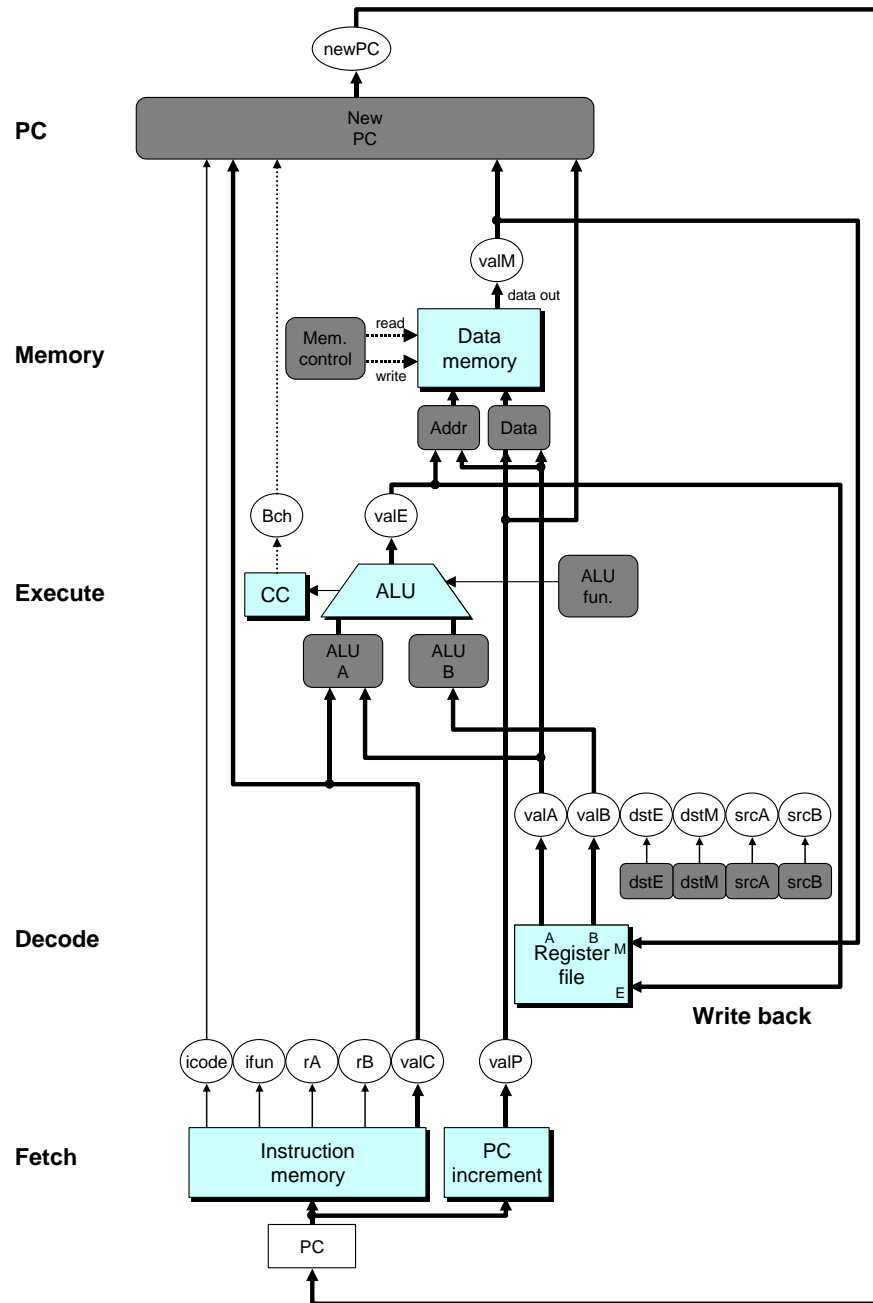
- Update program counter



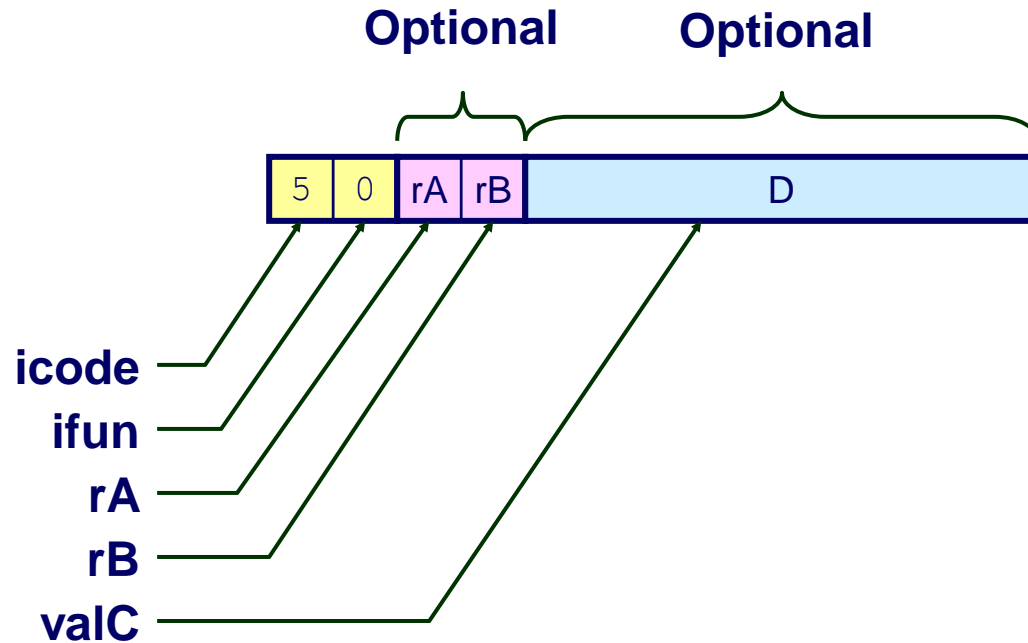
SEQ Hardware

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



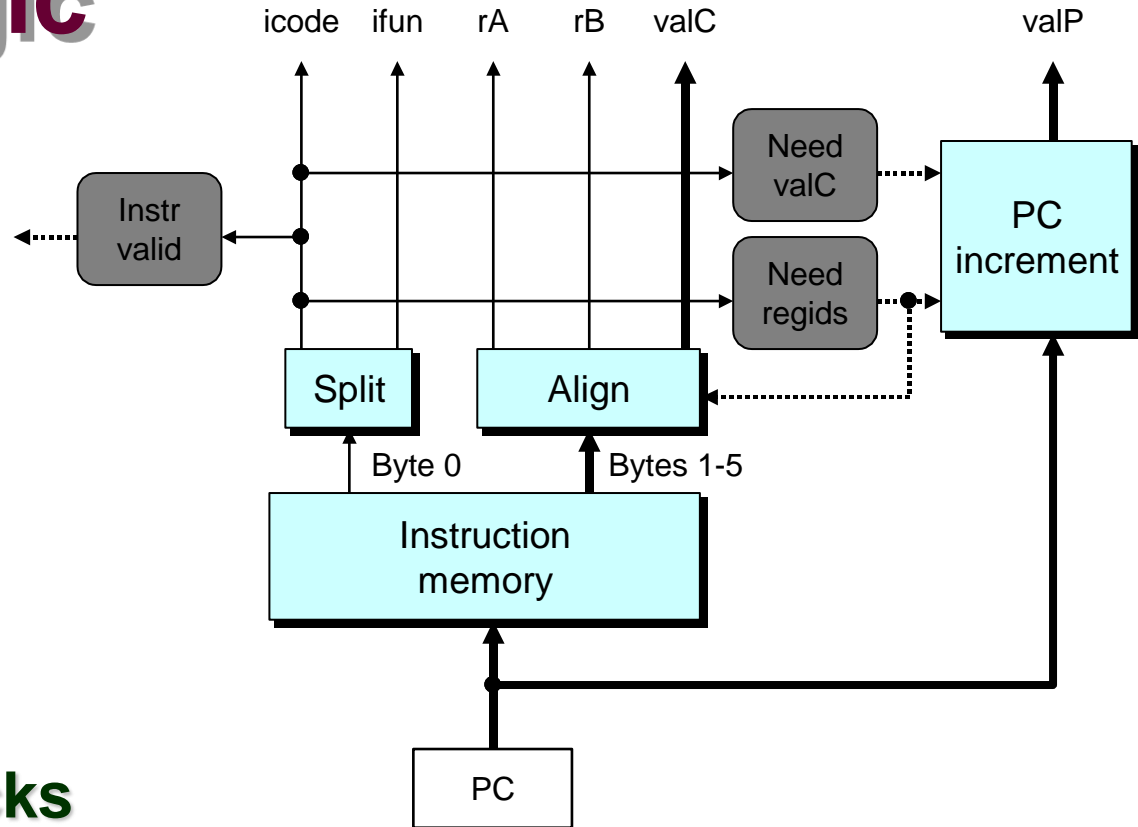
Instruction Decoding



Instruction Format

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

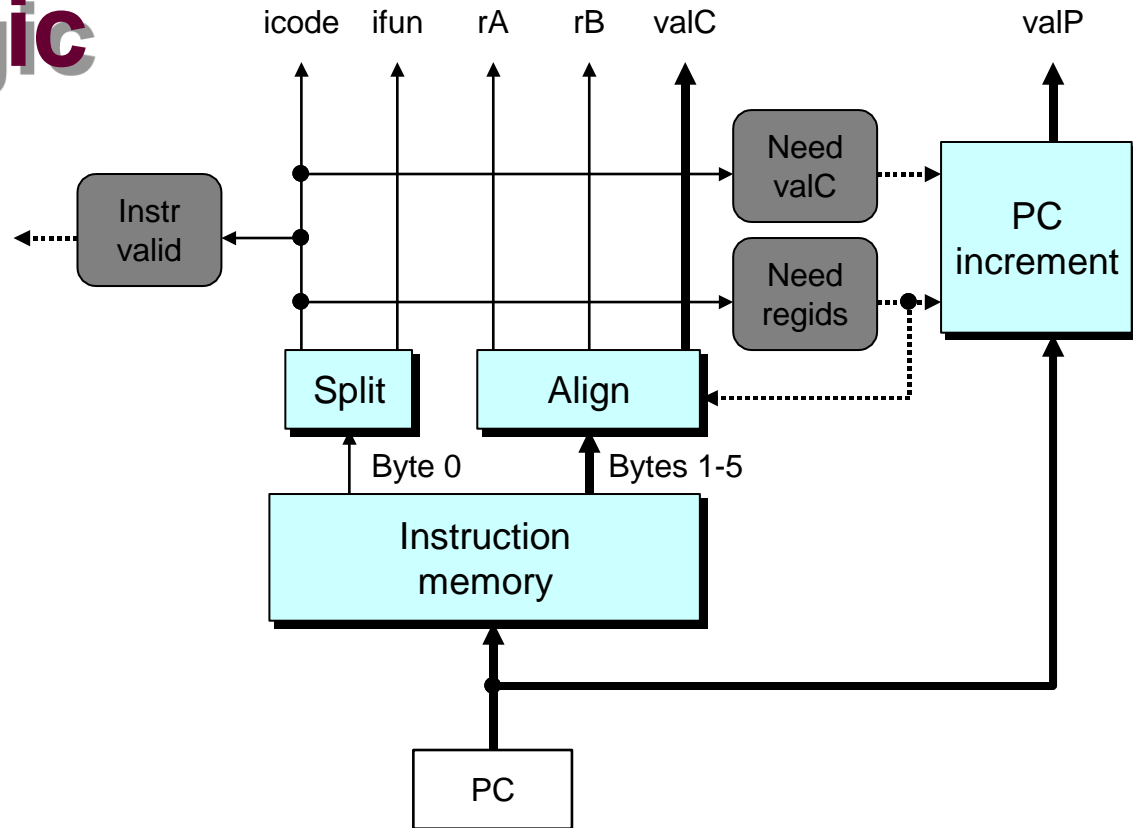
Fetch Logic



Predefined Blocks

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

Fetch Logic



Control Logic

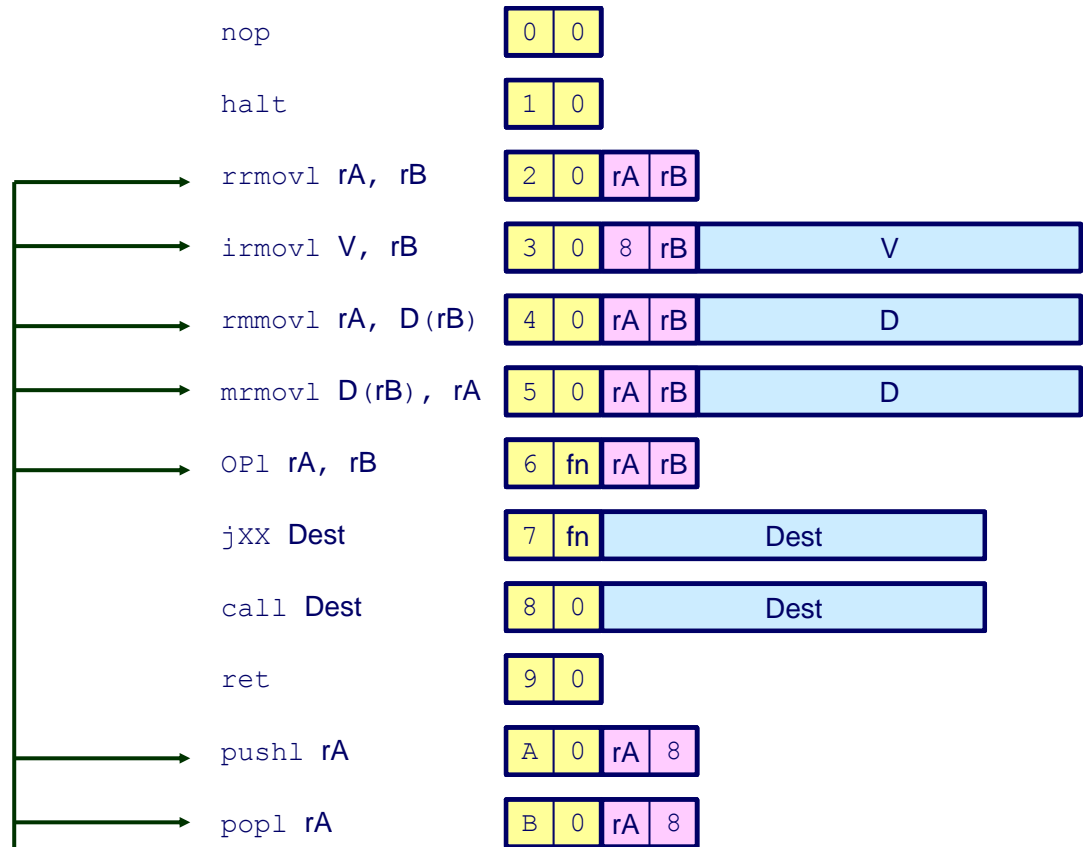
- **Instr. Valid:** Is this instruction valid?
- **Need regids:** Does this instruction have a register bytes?
- **Need valC:** Does this instruction have a constant word?

Fetch Control Logic

```
bool need_valC =
    icode in {
        IIRMOVL,
        IRMMOVL, IMRMOVL,
        IJXX, ICALL };
```

```
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 };
```



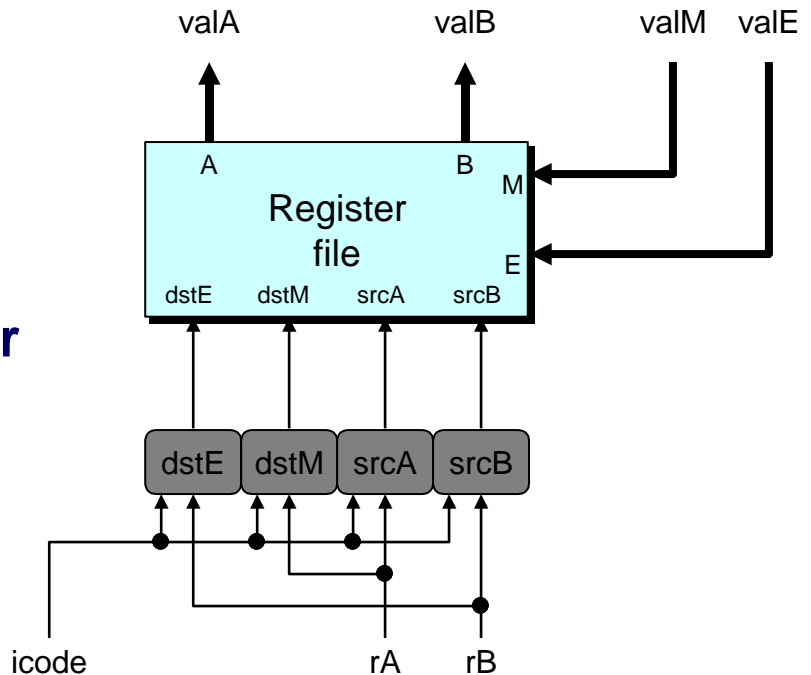
Decode Logic

Register File

- Read ports A, B
- Write ports E, M
- Addresses are register IDs or 8 (no access)

Control Logic

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



A Source

	OPl rA, rB	
Decode	valA ← R[rA]	Read operand A
	rmmovl rA, D(rB)	
Decode	valA ← R[rA]	Read operand A
	popl rA	
Decode	valA ← R[%esp]	Read stack pointer
	jXX Dest	
Decode		No operand
	call Dest	
Decode		No operand
	ret	
Decode	valA ← R[%esp]	Read stack pointer

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

E Destination

	OPl rA, rB	
Write-back	R[rB] ← valE	Write back result
	rmmovl rA, D(rB)	
Write-back		None
	popl rA	
Write-back	R[%esp] ← valE	Update stack pointer
	jXX Dest	
Write-back		None
	call Dest	
Write-back	R[%esp] ← valE	Update stack pointer
	ret	
Write-back	R[%esp] ← valE	Update stack pointer

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

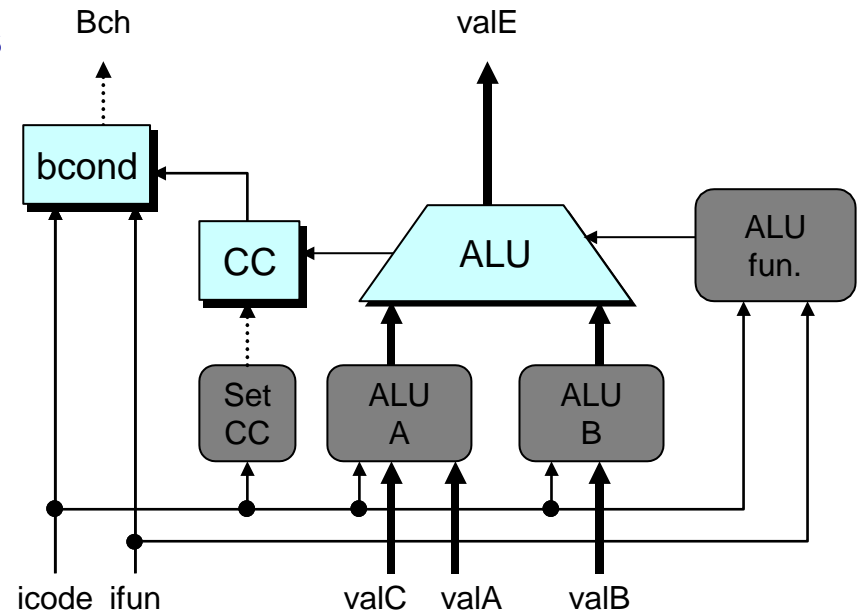
Execute Logic

Units

- **ALU**
 - Implements 4 required functions
 - Generates condition code values
- **CC**
 - Register with 3 condition code bits
- **bcond**
 - Computes branch 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	$valE \leftarrow valB \text{ OP } valA$	Perform ALU operation
	rmmovl rA, D(rB)	
Execute	$valE \leftarrow valB + valC$	Compute effective address
	popl rA	
Execute	$valE \leftarrow valB + 4$	Increment stack pointer
	jXX Dest	
Execute		No operation
	call Dest	
Execute	$valE \leftarrow valB + -4$	Decrement stack pointer
	ret	
Execute	$valE \leftarrow valB + 4$	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	$valE \leftarrow valB \text{ OP } valA$	Perform ALU operation
	rmmovl rA, D(rB)	
Execute	$valE \leftarrow valB + valC$	Compute effective address
	popl rA	
Execute	$valE \leftarrow valB + 4$	Increment stack pointer
	jXX Dest	
Execute		No operation
	call Dest	
Execute	$valE \leftarrow valB + -4$	Decrement stack pointer
	ret	
Execute	$valE \leftarrow valB + 4$	Increment stack pointer

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

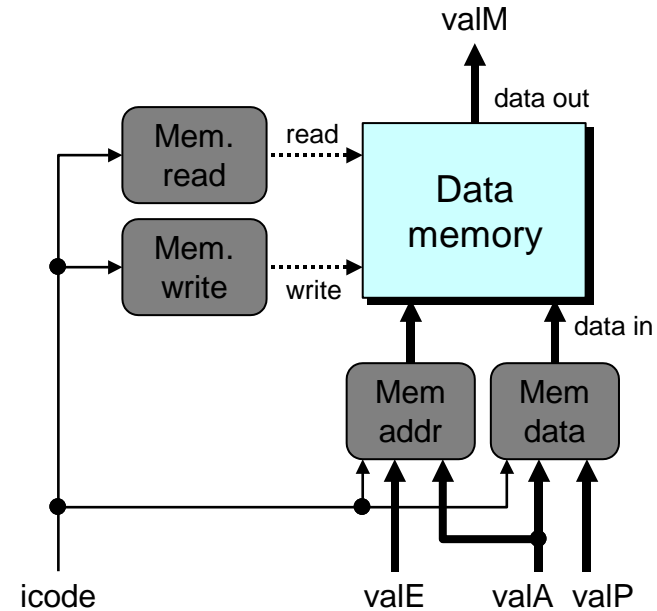
Memory Logic

Memory

- Reads or writes memory word

Control Logic

- Mem. read: should word be read?
- Mem. write: should word be written?
- Mem. addr.: Select address
- Mem. data.: Select data



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

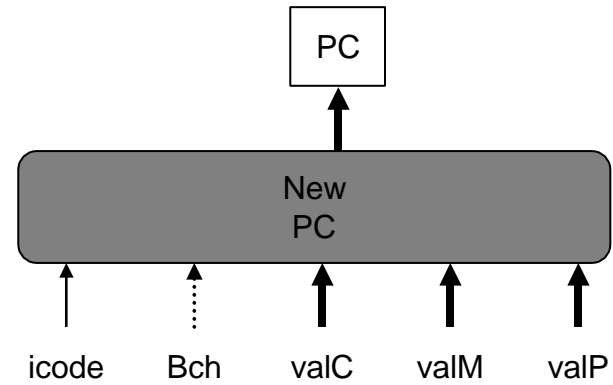
	OPl rA, rB	
Memory		No operation
	rmmovl rA, D(rB)	
Memory	$M_4[\text{valE}] \leftarrow \text{valA}$	Write value to memory
	popl rA	
Memory	$\text{valM} \leftarrow M_4[\text{valA}]$	Read from stack
	jXX Dest	
Memory		No operation
	call Dest	
Memory	$M_4[\text{valE}] \leftarrow \text{valP}$	Write return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

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

PC Update Logic

New PC

- Select next value of PC



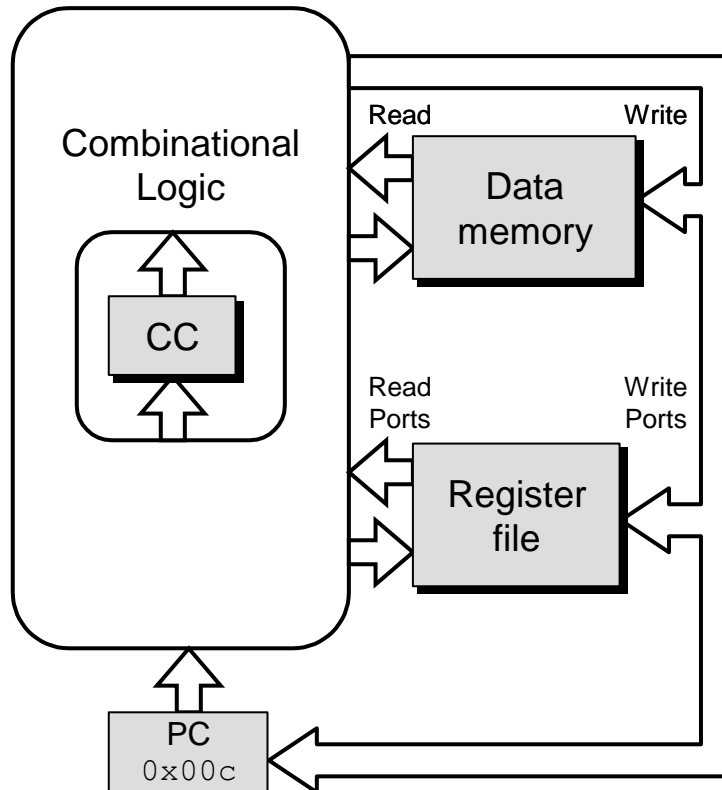
PC Update

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

SEQ Operation

State

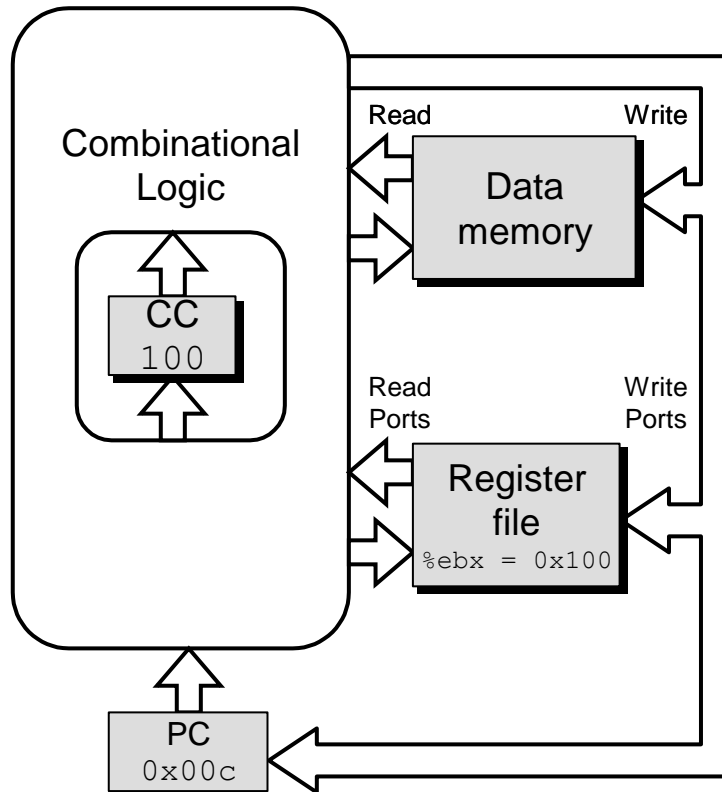
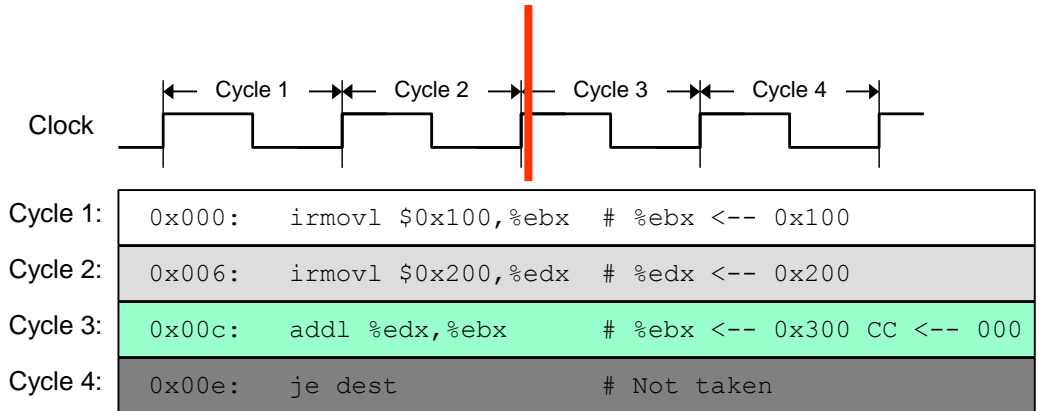


- PC register
 - Cond. Code register
 - Data memory
 - Register file
- All updated as clock rises*

Combinational Logic

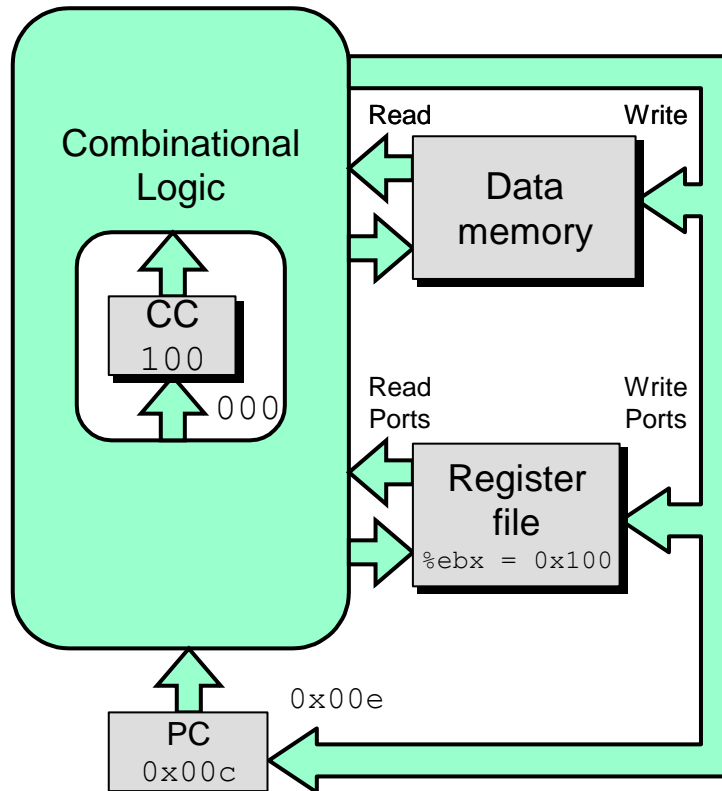
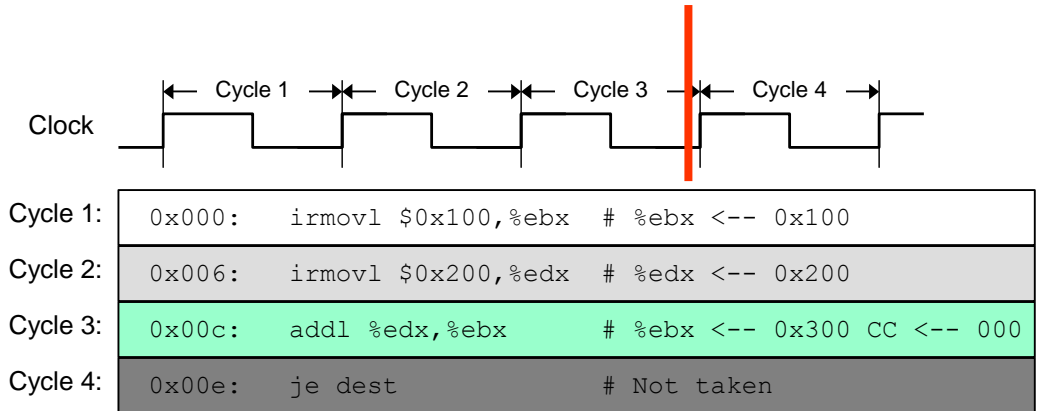
- ALU
- Control logic
- Memory reads
 - Instruction memory
 - Register file
 - Data memory

SEQ Operation #2



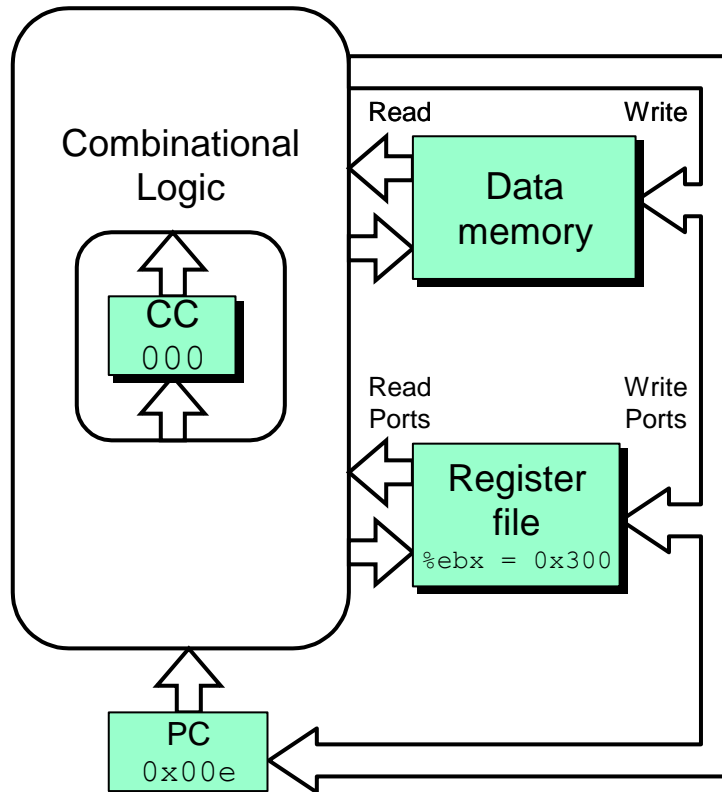
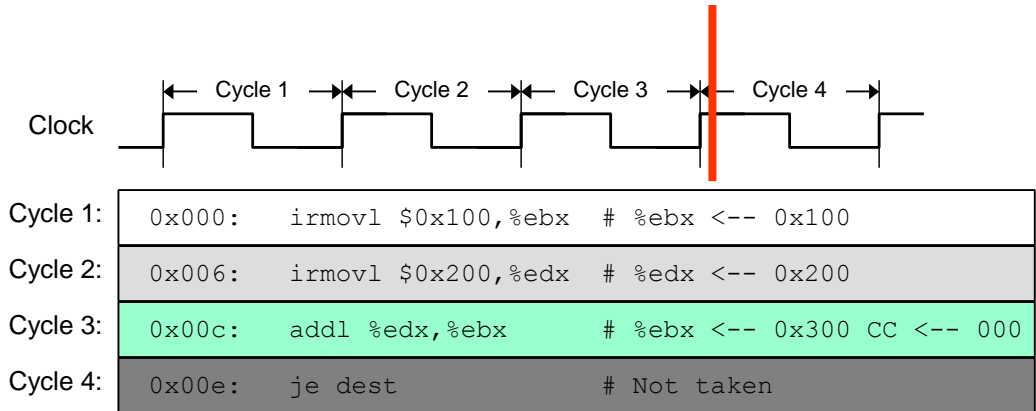
- state set according to second `irmovl` instruction
- combinational logic starting to react to state changes

SEQ Operation #3



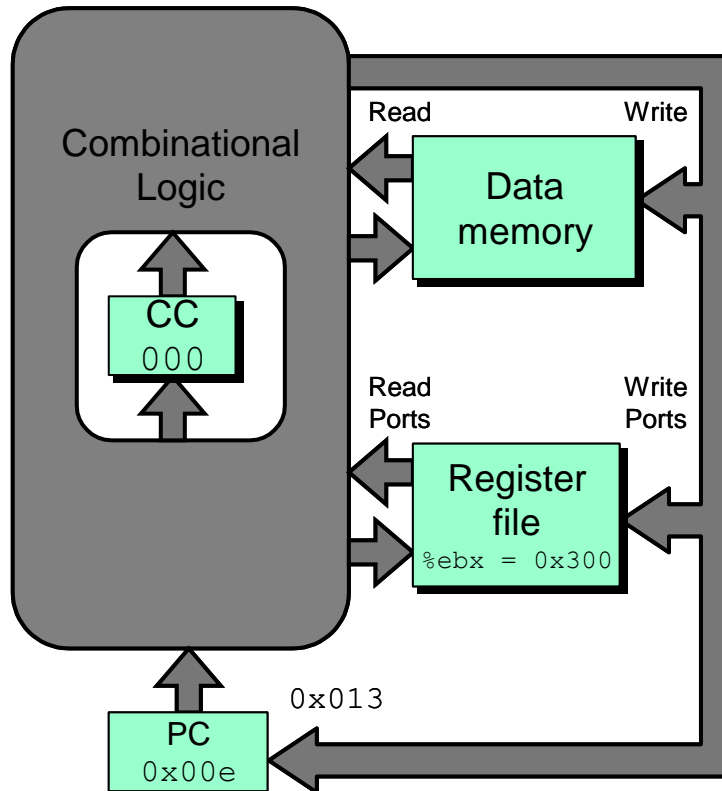
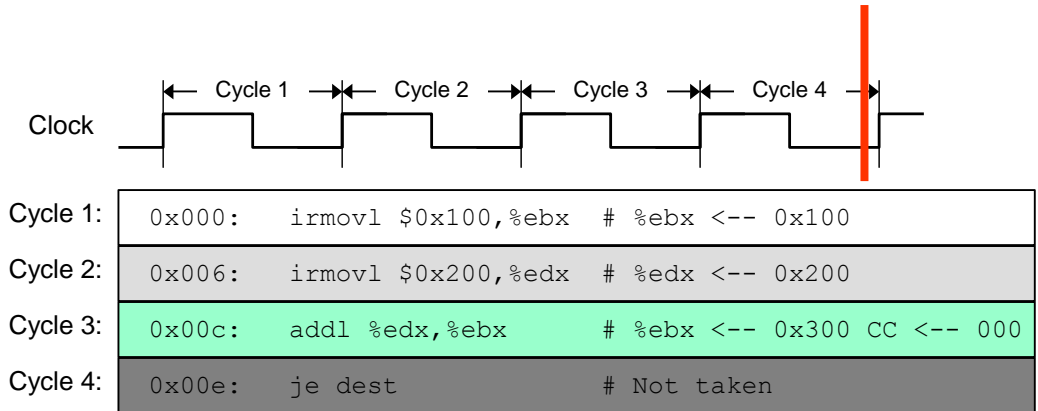
- state set according to second `irmovl` instruction
- combinational logic generates results for `addl` instruction

SEQ Operation #4



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

SEQ Operation #5



- 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

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