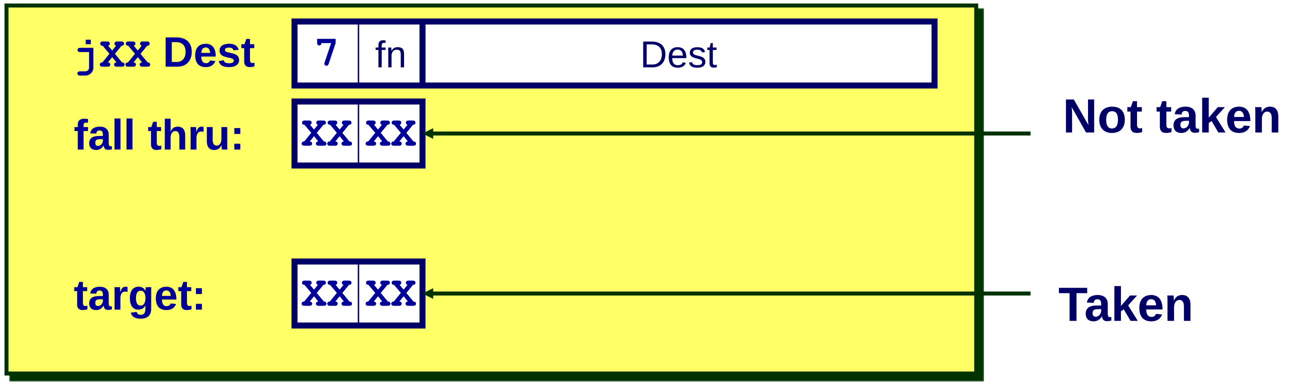
Systems I

Datapath Design II

Topics

- Control flow instructions
- Hardware for sequential machine (SEQ)

Executing Jumps



Fetch

- Read 5 bytes
- Increment PC by 5

Decode

- Do nothing

Execute

- Determine whether to take branch based on jump condition and condition codes

Memory

- Do nothing

Write back

- Do nothing

PC Update

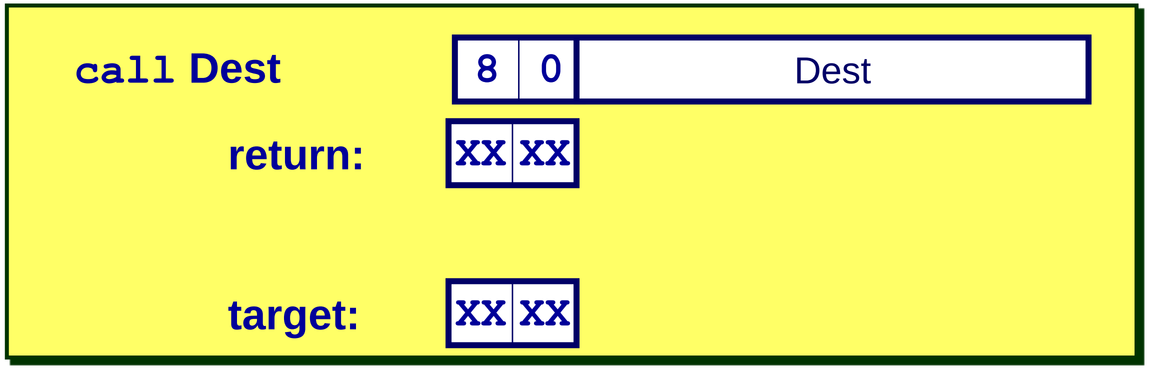
- Set PC to Dest if branch taken or to incremented PC if not branch

Stage Computation: Jumps

	jXX Dest	
Fetch	icode:ifun $\leftarrow$ M_1[PC]	Read instruction byte
	valC $\leftarrow$ M_4[PC+1]	Read destination address
	valP $\leftarrow$ PC+5	Fall through address
Decode		
Execute	Bch $\leftarrow$ Cond(CC,ifun)	Take branch?
Memory		
Write back		
PC update	PC $\leftarrow$ Bch ? valC : valP	Update PC

- Compute both addresses
- Choose based on setting of condition codes and branch condition

Executing `call`



Fetch

- Read 5 bytes
- Increment PC by 5

Decode

- Read stack pointer

Execute

- Decrement stack pointer by 4

Memory

- Write incremented PC to new value of stack pointer

Write back

- Update stack pointer

PC Update

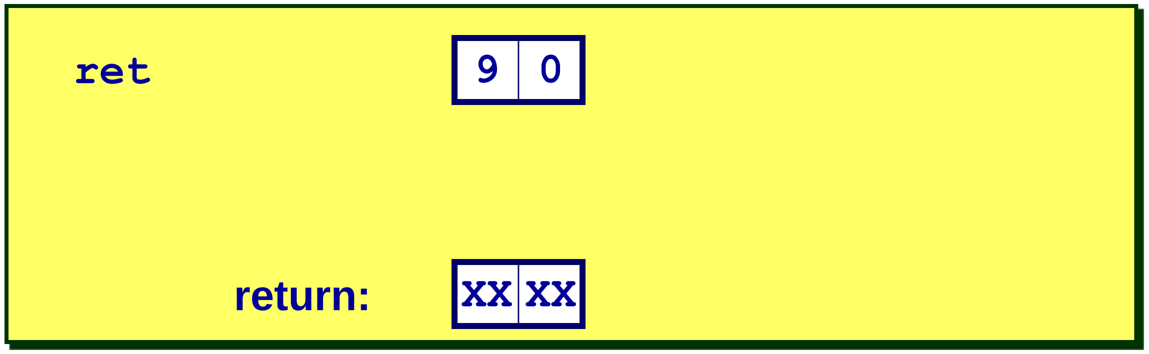
- Set PC to Dest

Stage Computation: `call`

	`call` Dest	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_4[PC+1]	Read destination address
	valP ← PC+5	Compute return point
Decode		
	valB ← R[`%esp`]	Read stack pointer
Execute	valE ← valB + –4	Decrement stack pointer
Memory	M_4[valE] ← valP	Write return value on stack
Write back	R[`%esp`] ← valE	Update stack pointer
PC update	PC ← valC	Set PC to destination

- Use ALU to decrement stack pointer
- Store incremented PC

Executing `ret`



Fetch

- Read 1 byte

Decode

- Read stack pointer

Execute

- Increment stack pointer by 4

Memory

- Read return address from old stack pointer

Write back

- Update stack pointer

PC Update

- Set PC to return address

Stage Computation: `ret`

	`ret`	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
Decode	valA ← R[`%esp`]	Read operand stack pointer
	valB ← R[`%esp`]	Read operand stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M_4[valA]	Read return address
Write back	R[`%esp`] ← 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

		OPl 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 condition code register
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 $\leftarrow$ M_1[PC]	Read instruction byte
	rA,rB		[Read register byte]
	valC	valC $\leftarrow$ M_4[PC+1]	Read constant word
	valP	valP $\leftarrow$ PC+5	Compute next PC
Decode	valA, srcA		[Read operand A]
	valB, srcB	valB $\leftarrow$ R[`%esp`]	Read operand B
Execute	valE	valE $\leftarrow$ valB + –4	Perform ALU operation
	Cond code		[Set condition code reg.]
Memory	valM	M_4[valE] $\leftarrow$ valP	[Memory read/write]
Write back	dstE	R[`%esp`] $\leftarrow$ 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
- Bch Branch flag

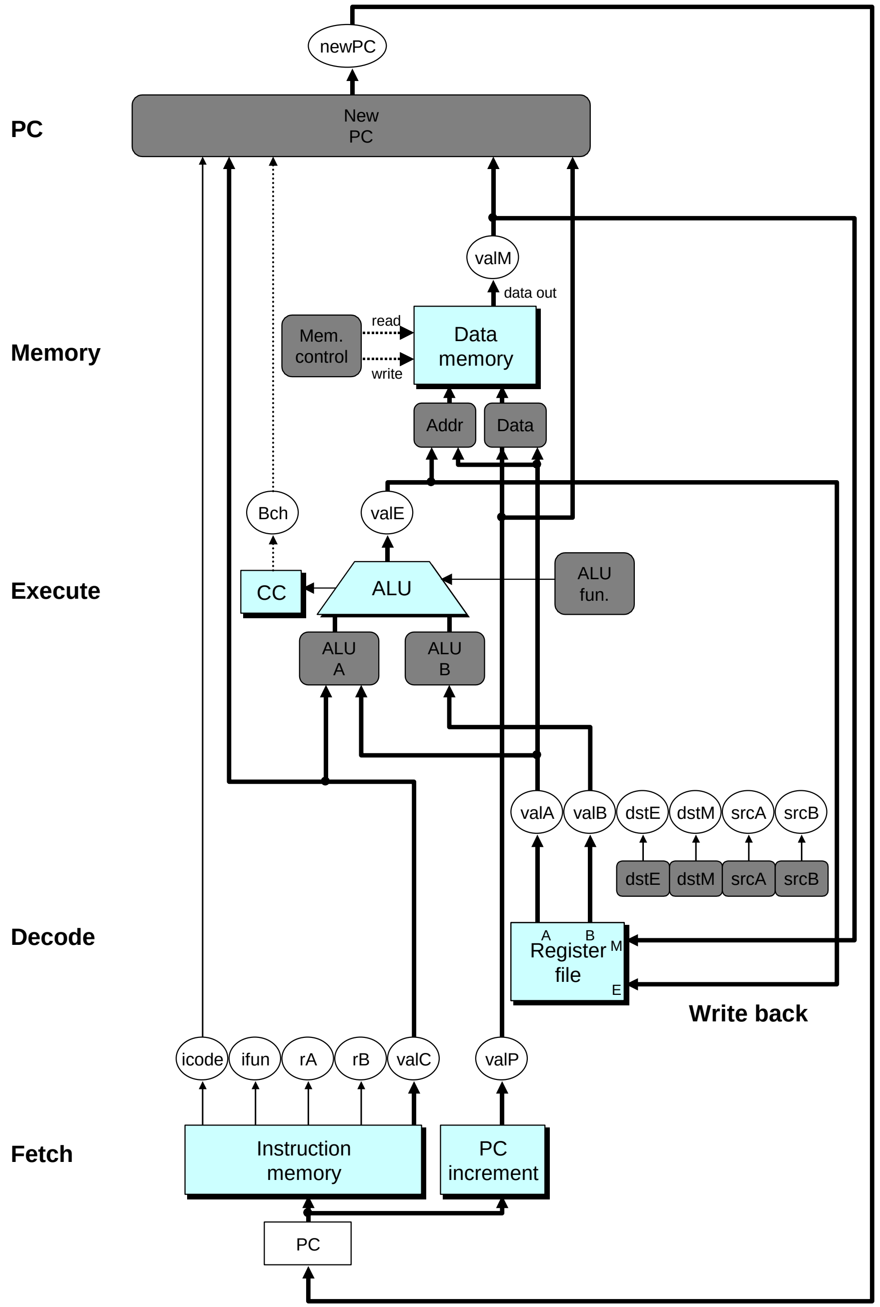
Memory

- valM Value from memory

SEQ Hardware

Key

- **Blue boxes: predesigned hardware blocks**
  - **E.g., memories, ALU**
- **Gray boxes: control logic**
  - **Describe in HCL**
- **White ovals: labels for signals**
- **Thick lines: 32-bit word values**
- **Thin lines: 4-8 bit values**
- **Dotted lines: 1-bit values**



Summary

Today

- Control flow instructions
- Hardware for sequential machine (SEQ)

Next time

- Control logic for instruction execution
- Timing and clocking

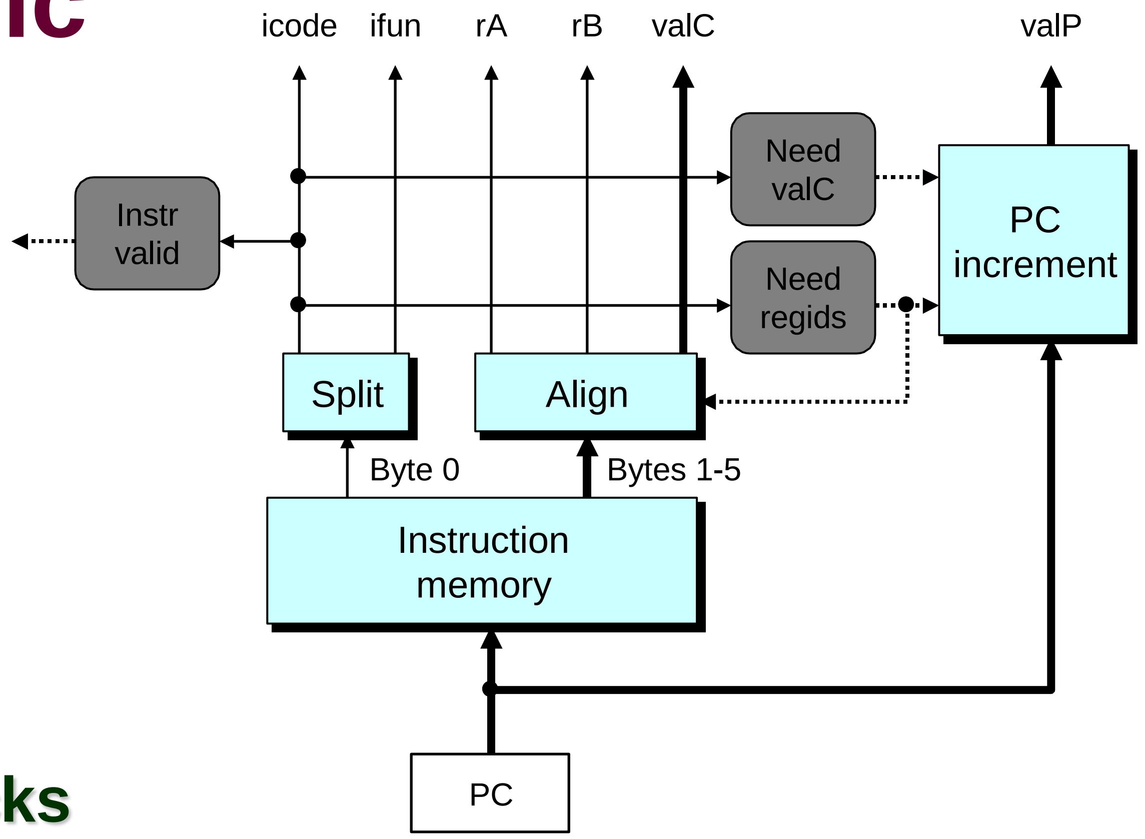
Systems I

Datapath Design III

Topics

- Control logic for instruction execution
- Timing and clocking

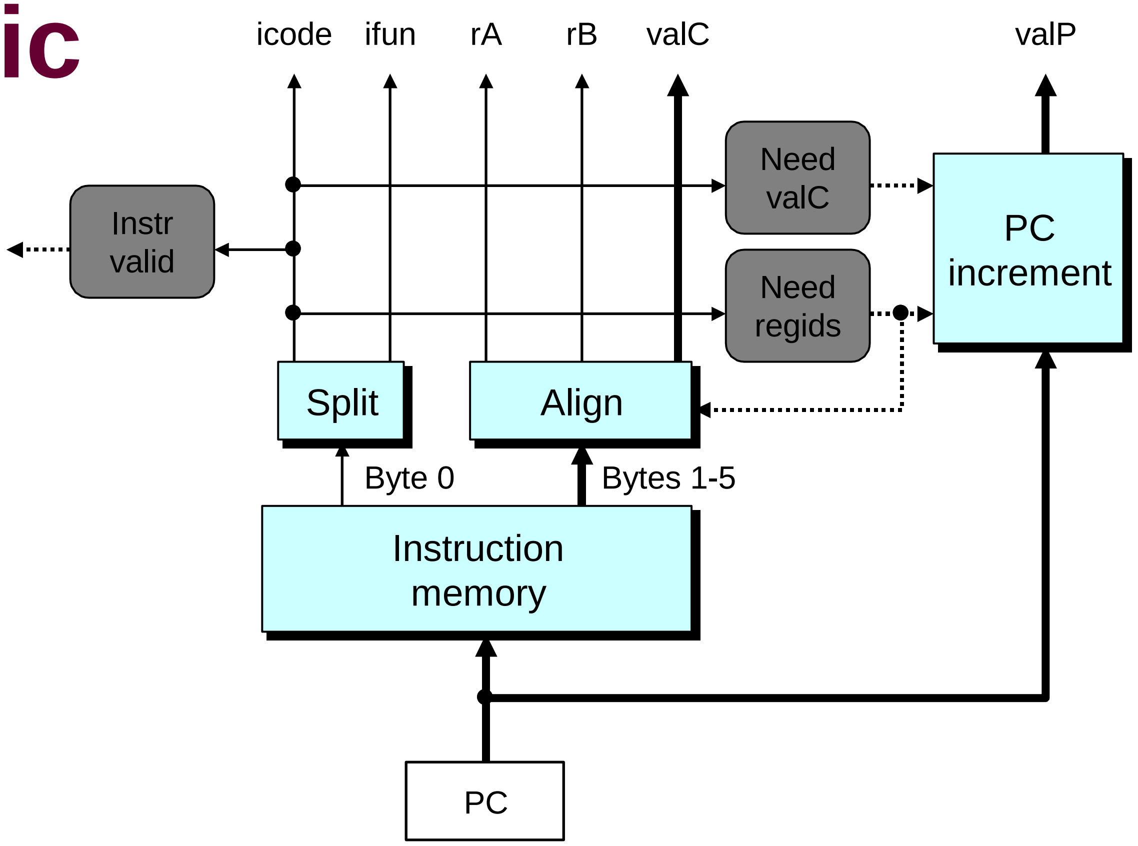
Fetch Logic



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

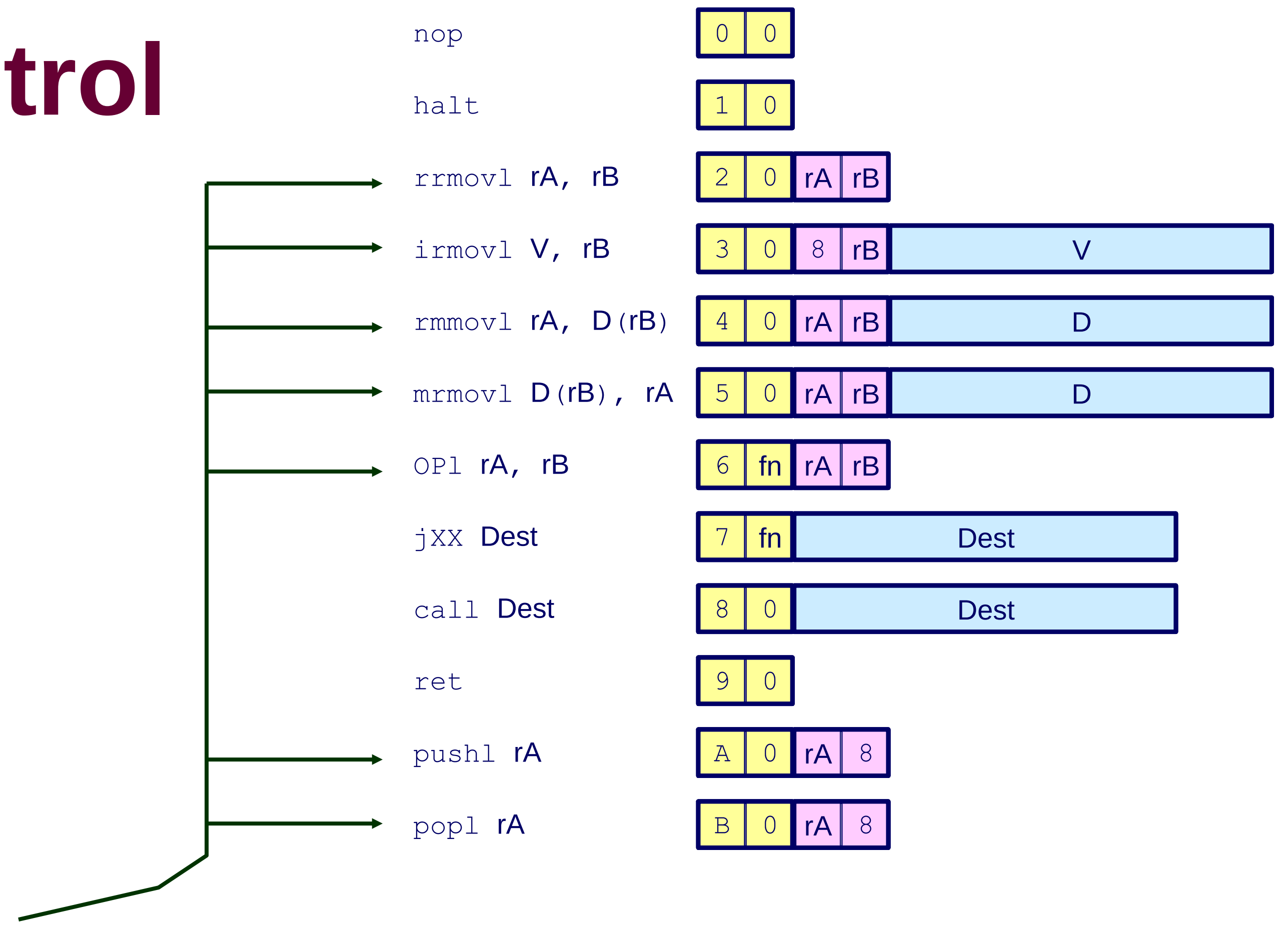
Fetch Logic



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_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 };
```

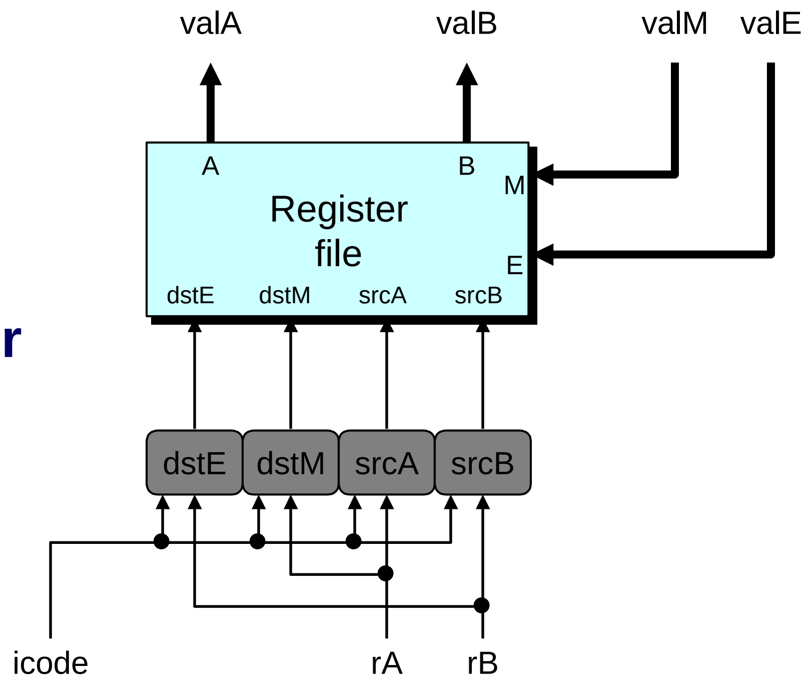
Decode Logic

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
- dstA, dstB: 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
];
```

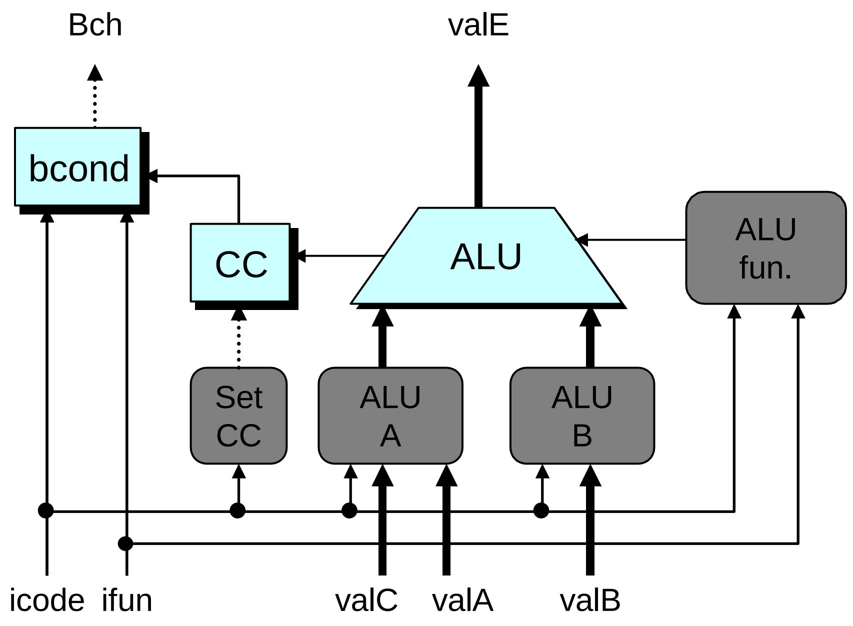
Execute Logic

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 ← valB OP valA	Perform ALU operation

	`rmmovl` rA, D(rB)	
Execute	valE ← valB + valC	Compute effective address

	`popl` rA	
Execute	valE ← valB + 4	Increment stack pointer

	jXX Dest	
Execute		No operation

	`call` Dest	
Execute	valE ← valB + –4	Decrement stack pointer

	`ret`	
Execute	valE ← 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 ← valB OP valA	Perform ALU operation
	`rmmovl` rA, D(rB)	
Execute	valE ← valB + valC	Compute effective address
	`popl` rA	
Execute	valE ← valB + 4	Increment stack pointer
	jXX Dest	
Execute		No operation
	`call` Dest	
Execute	valE ← valB + –4	Decrement stack pointer
	`ret`	
Execute	valE ← valB + 4	Increment stack pointer

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

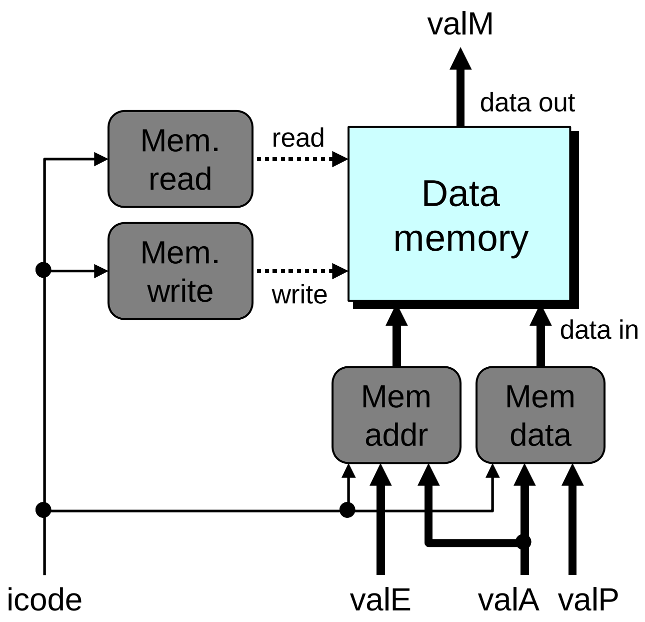
Memory Logic

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[valE] ← valA	Write value to memory

	popl rA	
Memory	valM ← M_4[valA]	Read from stack

	jXX Dest	
Memory		No operation

	call Dest	
Memory	M_4[valE] ← valP	Write return value on stack

	ret	
Memory	valM ← M_4[valA]	Read return address

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

Memory Read

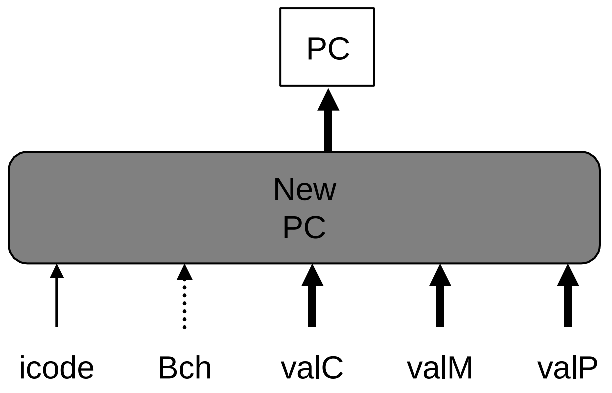
	OPl rA, rB	
Memory		No operation
	`rmmovl` rA, D(rB)	
Memory	M_4[valE] $\leftarrow$ valA	Write value to memory
	`popl` rA	
Memory	valM $\leftarrow$ M_4[valA]	Read from stack
	jXX Dest	
Memory		No operation
	`call` Dest	
Memory	M_4[valE] $\leftarrow$ valP	Write return value on stack
	`ret`	
Memory	valM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

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

PC Update Logic

New PC

- Select next value of PC



PC Update

	OPl rA, rB	
PC update	PC ← valP	Update PC

	`rmmovl` rA, D(rB)	
PC update	PC ← valP	Update PC

	`popl` rA	
PC update	PC ← valP	Update PC

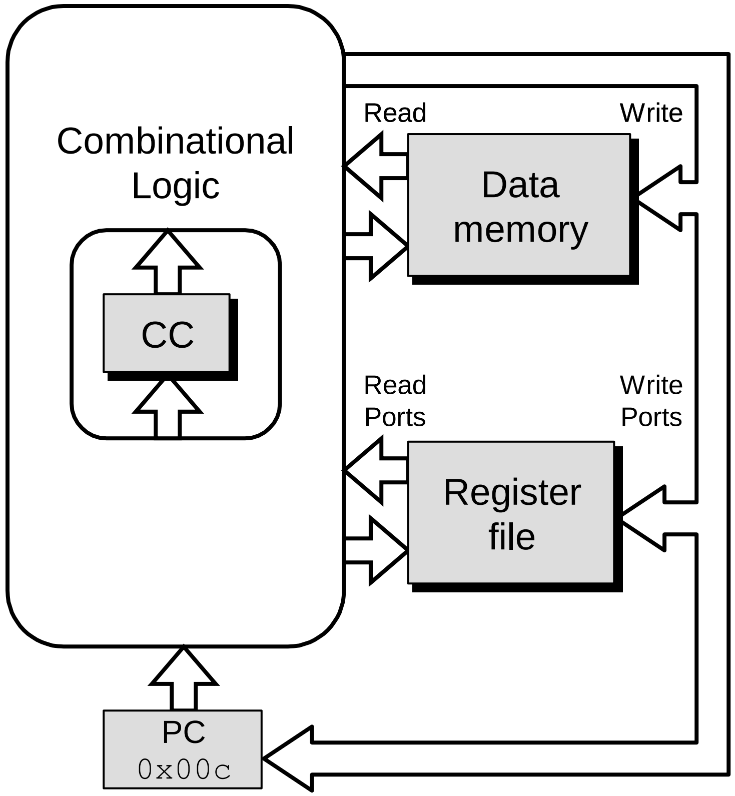
	jXX Dest	
PC update	PC ← Bch ? valC : valP	Update PC

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

SEQ Operation



State

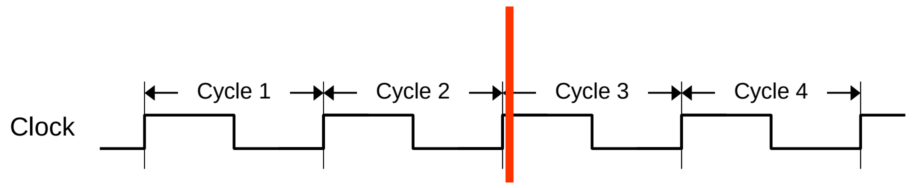
- PC register
- Cond. Code register
- Data memory
- Register file

All updated as clock rises

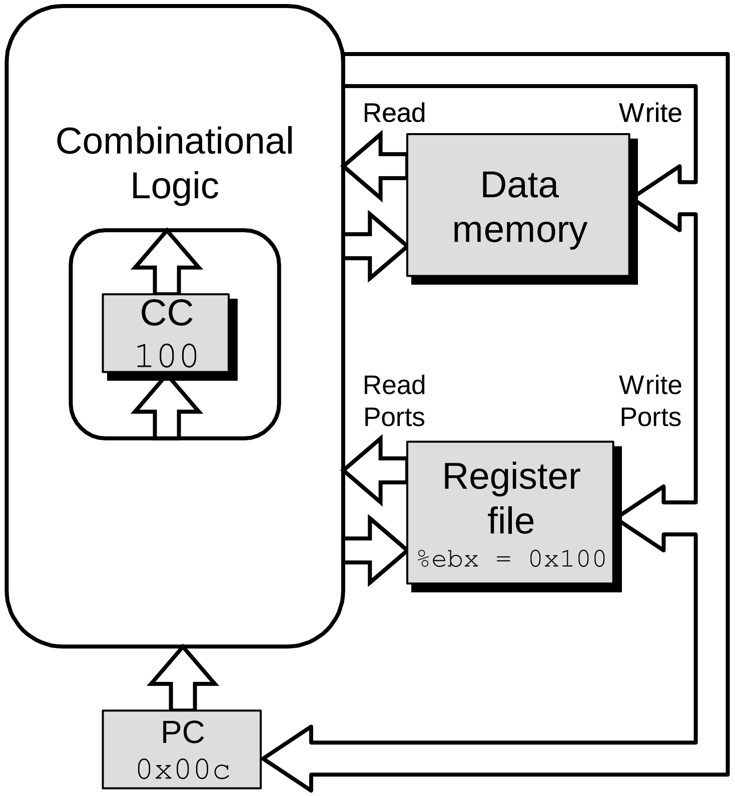
Combinational Logic

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

SEQ Operation #2

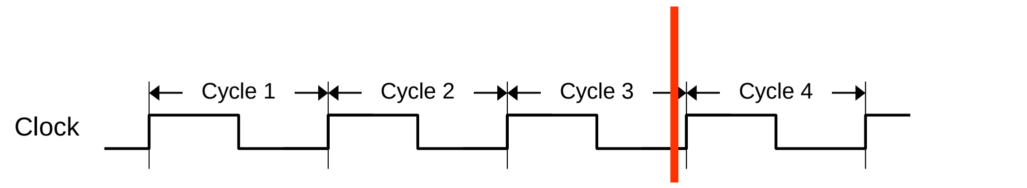


```
Cycle 1:  0x000:   irmovl $0x100,%ebx   # %ebx <-- 0x100
Cycle 2:  0x006:   irmovl $0x200,%edx   # %edx <-- 0x200
Cycle 3:  0x00c:   addl %edx,%ebx       # %ebx <-- 0x300 CC <-- 000
Cycle 4:  0x00e:   je dest              # Not taken
```

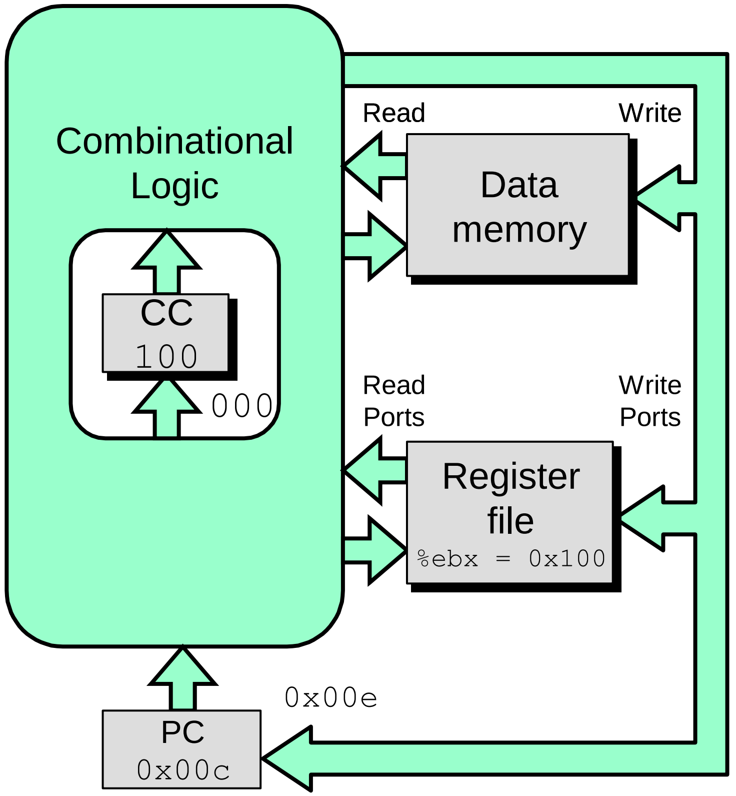


- **state set according to second `irmovl` instruction**
- **combinational logic starting to react to state changes**

SEQ Operation #3

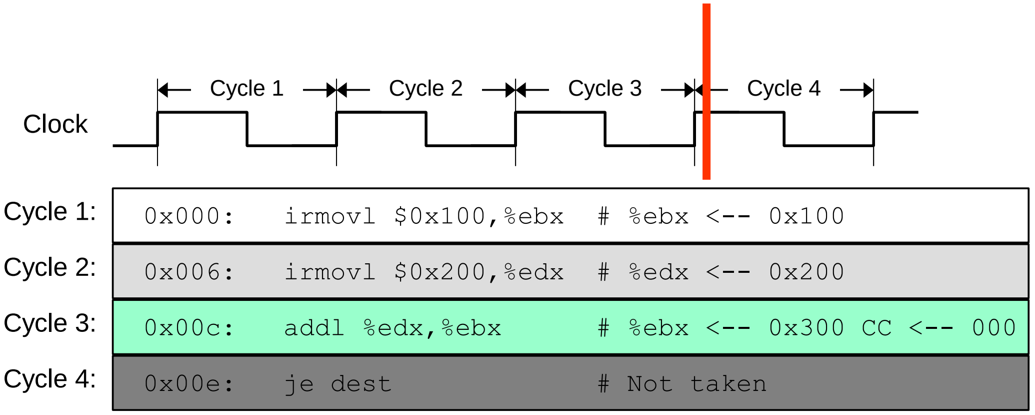


```
Cycle 1:  0x000:  irmovl $0x100,%ebx   # %ebx <-- 0x100
Cycle 2:  0x006:  irmovl $0x200,%edx   # %edx <-- 0x200
Cycle 3:  0x00c:  addl %edx,%ebx       # %ebx <-- 0x300 CC <-- 000
Cycle 4:  0x00e:  je dest              # Not taken
```

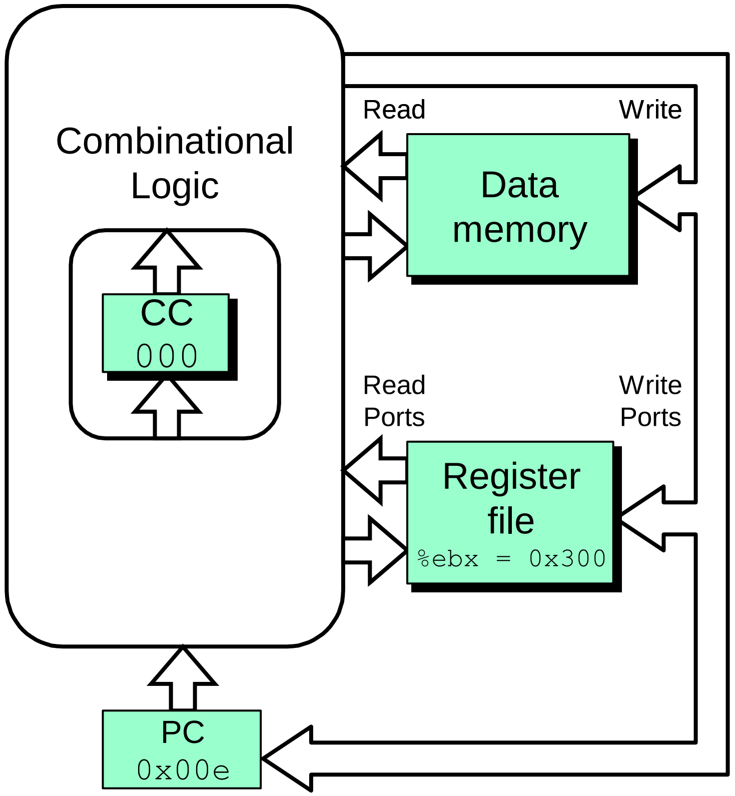


- **state set according to second `irmovl` instruction**
- **combinational logic generates results for `addl` instruction**

SEQ Operation #4

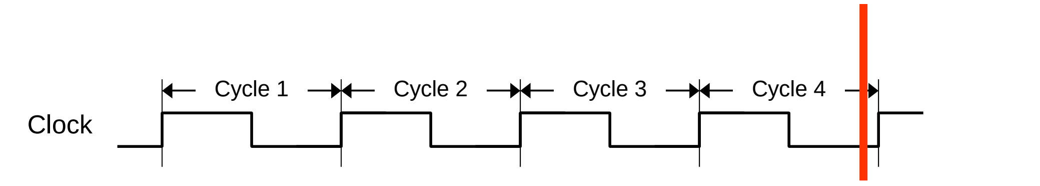


```
Cycle 1:  0x000:   irmovl $0x100,%ebx   # %ebx <-- 0x100
Cycle 2:  0x006:   irmovl $0x200,%edx   # %edx <-- 0x200
Cycle 3:  0x00c:   addl %edx,%ebx       # %ebx <-- 0x300 CC <-- 000
Cycle 4:  0x00e:   je dest              # Not taken
```

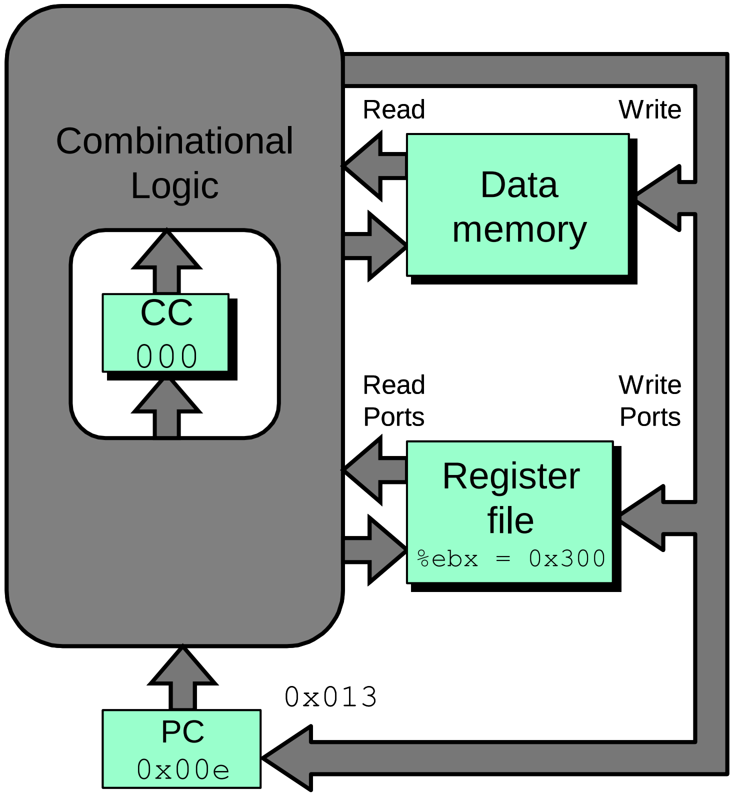


- state set according to `addl` instruction
- combinational logic starting to react to state changes

SEQ Operation #5



```
Cycle 1:   0x000:   irmovl $0x100,%ebx   # %ebx <-- 0x100
Cycle 2:   0x006:   irmovl $0x200,%edx   # %edx <-- 0x200
Cycle 3:   0x00c:   addl %edx,%ebx       # %ebx <-- 0x300 CC <-- 000
Cycle 4:   0x00e:   je dest              # Not taken
```



- **state set according to `addl` instruction**
- **combinational logic generates results for `je` instruction**

SEQ Summary

Implementation

- Express every instruction as series of simple steps
- Follow same general flow for each instruction type
- Assemble registers, memories, predesigned combinational blocks
- Connect with control logic

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