CS429: Computer Organization and Architecture

Datapath II

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: July 11, 2019 at 09:09

The ISA

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								

SEQ Stages

Fetch: Read instruction from instruction memory.

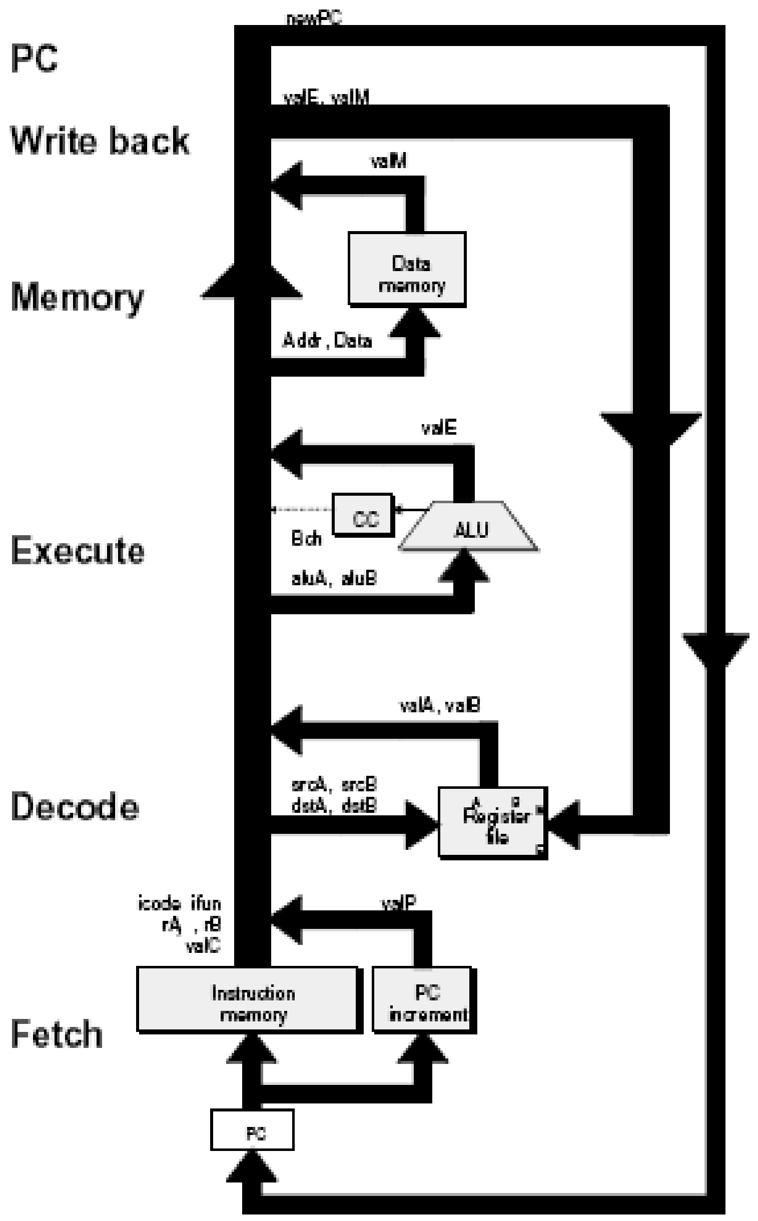
Decode: Read program registers

Execute: Compute value or address

Memory: Read or write back data.

Write Back: Write program registers.

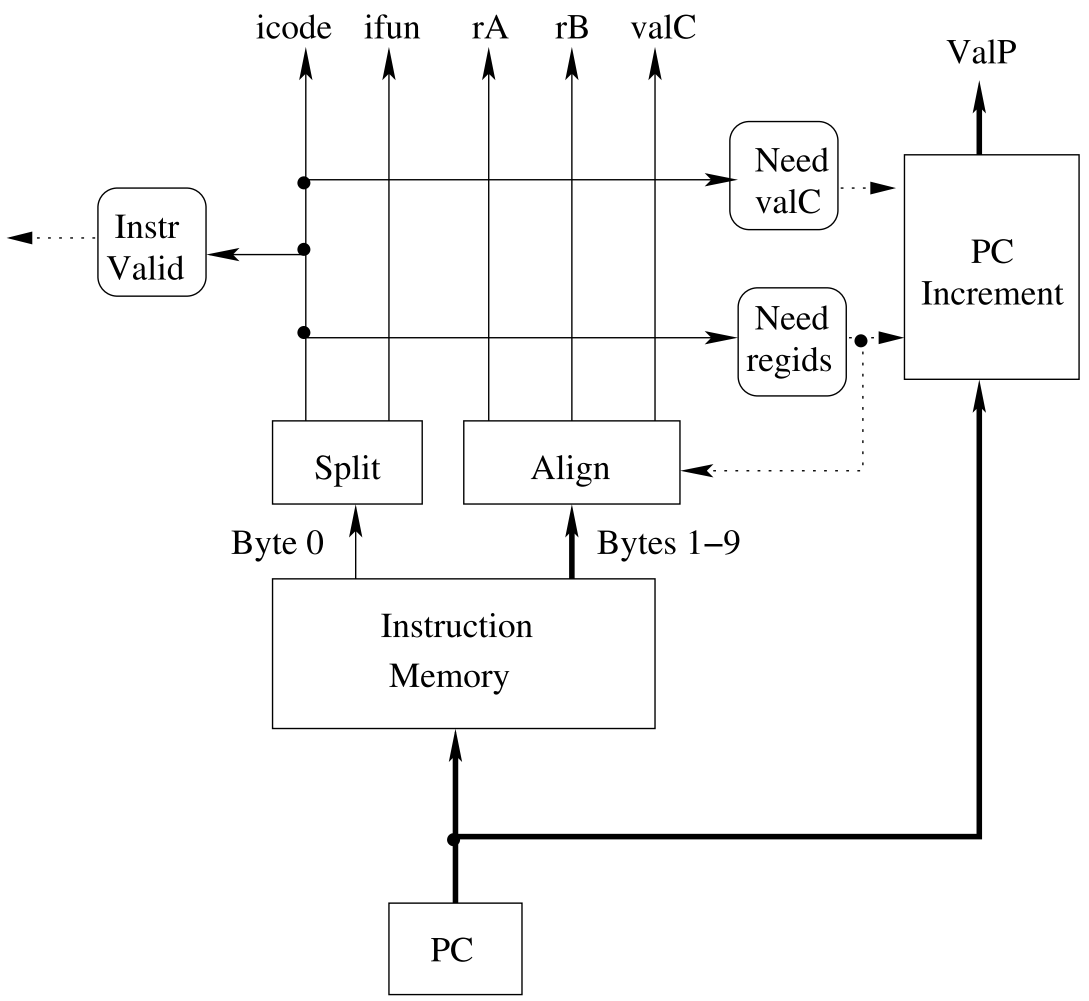
PC: Update the program counter.



Fetch Logic

Predefined Blocks

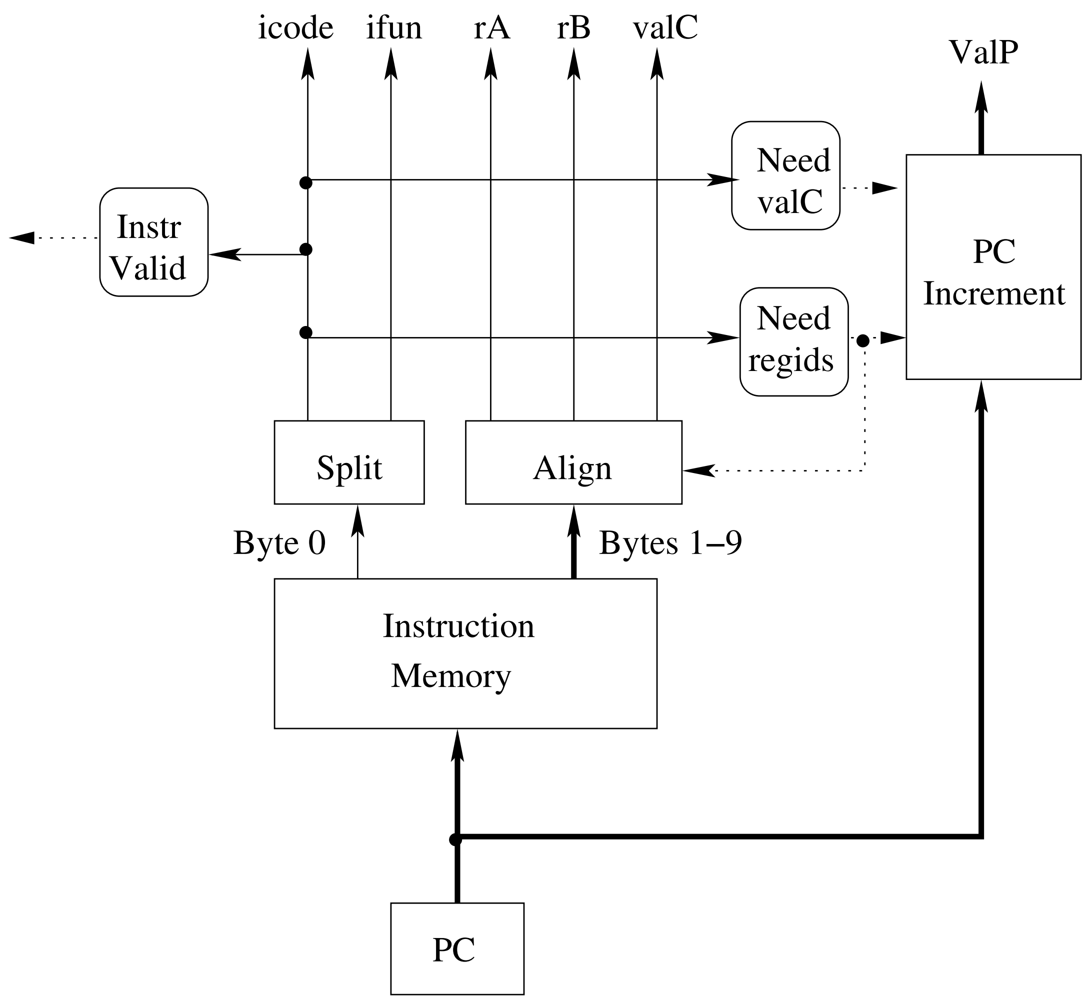
- **PC:** Register containing the PC.
- **Instruction memory:** Read 10 bytes (PC to PC+9).
- **Split:** Divide instruction byte into icode and ifun.
- **Align:** Get fields for rA, rB, and valC.



Fetch Logic

Control Logic

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



Fetch Control Logic

We can define how the various signals are computed using our HCL language:

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

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

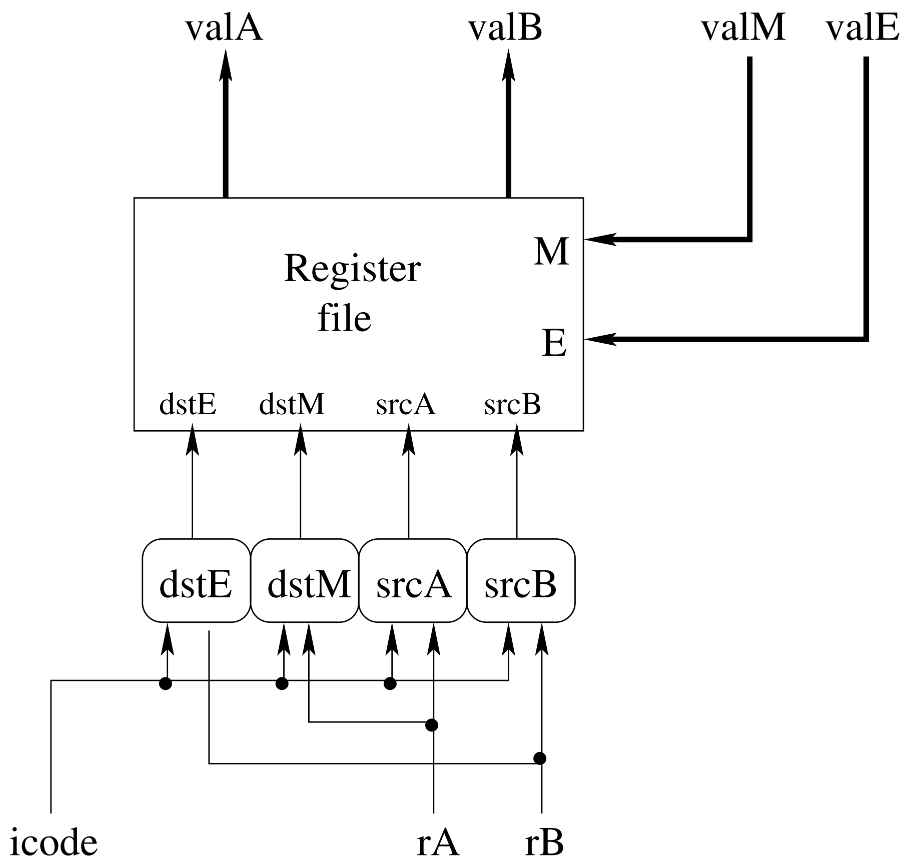
Decode Logic

Register File

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

Control Logic

- srA, srB: read port addresses
- dstA, dstB: write port addresses



Note that wires in the implementation have different semantics depending on the operation.

Source A

Where is register A coming from?

	`OPq 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 }: RESP;
   1: RNONE                           # Don't need register
];
```

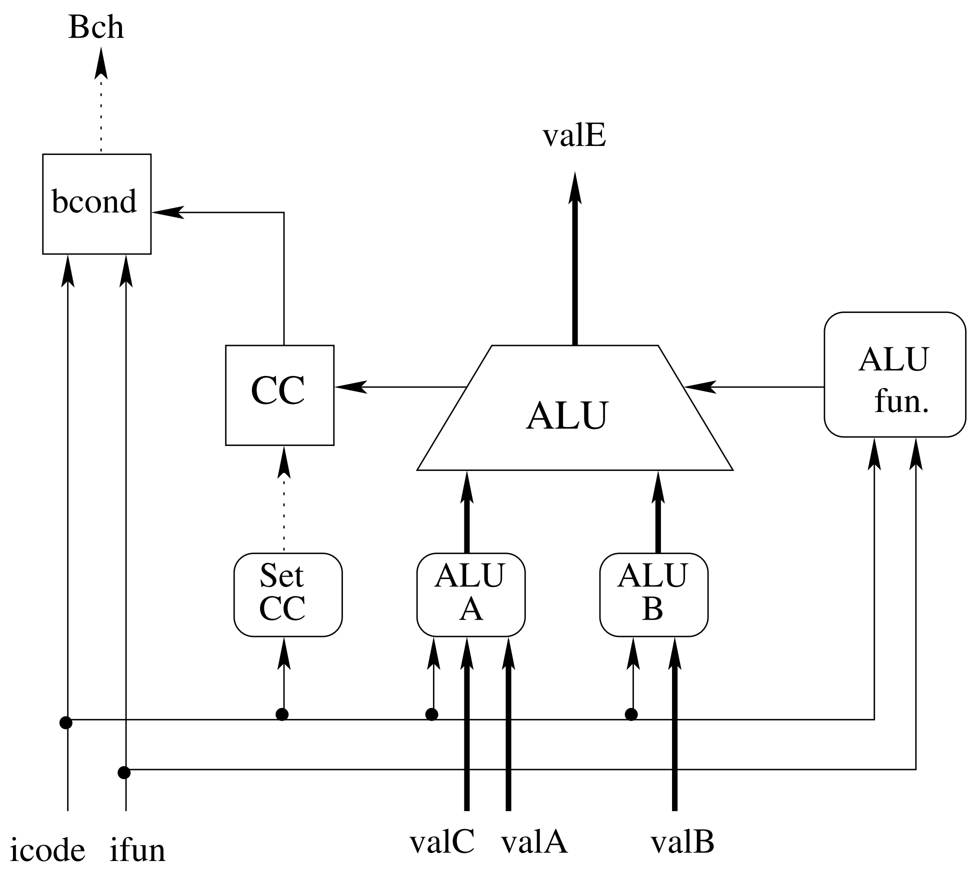
Execute Logic

Units

- ALU: Implements the 4 required functions, and 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

What is feeding the A input to the ALU?

	`OPq rA,rB`	
Execute	valE ← valB OP valA	Perform ALU operation

	`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 an ALU
];
```

ALU Operation

What function should the ALU perform?

	`OPq rA,rB`	
Execute	valE ← valB OP valA	Perform ALU operation (op)

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

	`popq rA`	
Execute	valE ← valB + 8	Increment stack pointer (add)

	`jXX Dest`	
Execute		No operation

	`call Dest`	
Execute	valE ← valB + -8	Decrement stack pointer (add)

	`ret`	
Execute	valE ← valB + 8	Increment stack pointer (add)

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

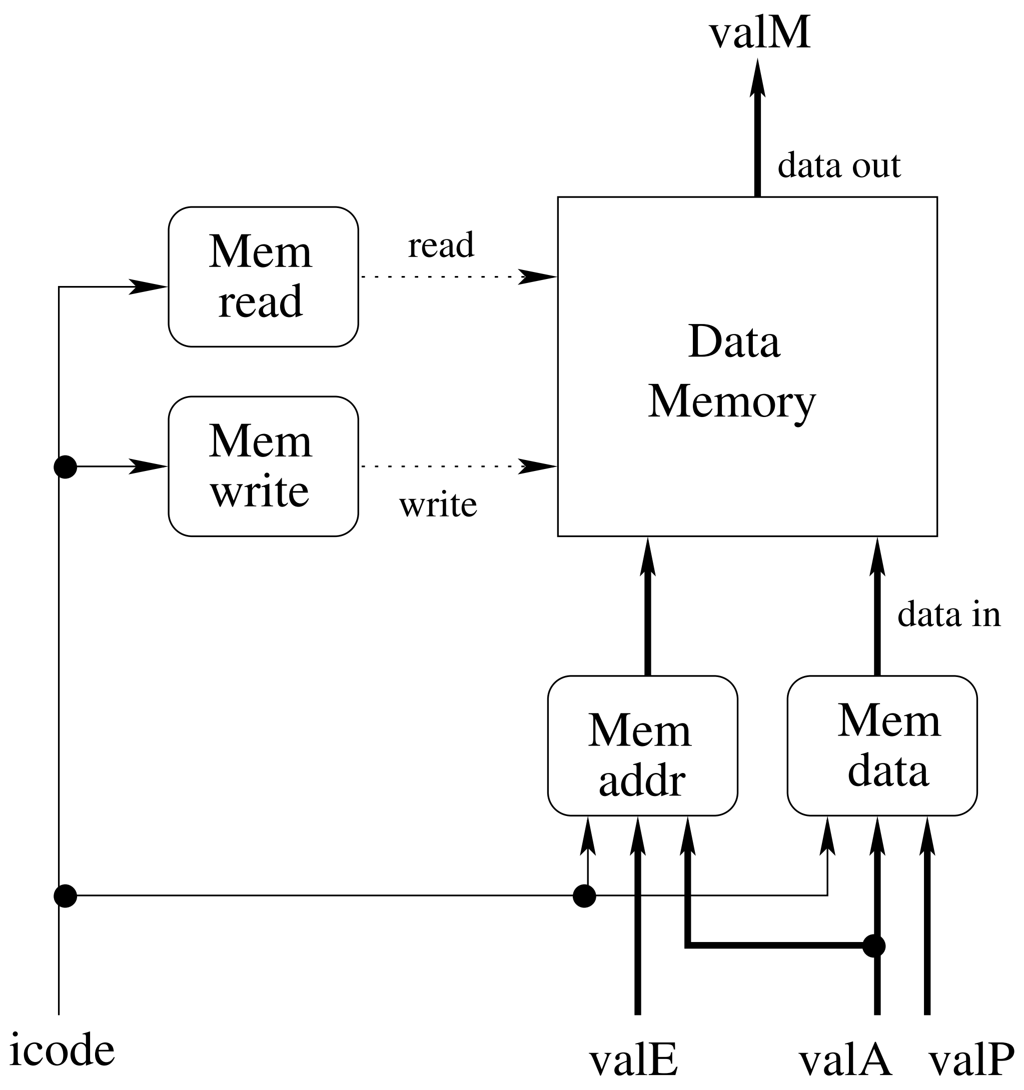
Memory Logic

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

What memory address is stored / loaded?

	OPq rA,rB	
Memory		No operation
	rmmovq rA,D(rB)	
Memory	M8[valE] ← valA	Write value to memory
	popq rA	
Memory	valM ← M8[valA]	Read from stack
	jXX Dest	
Memory		No operation
	call Dest	
Memory	M8[valE] ← valP	Write return value on stack
	ret	
Memory	valM ← M8[valA]	Increment stack pointer

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

Memory Read

For what instructions is memory read?

	`OPq rA,rB`	
Memory		No operation

	`rmmovq rA,D(rB)`	
Memory	M8[valE] $\leftarrow$ valA	Write value to memory

	`popq rA`	
Memory	valM $\leftarrow$ M8[valA]	Read from stack

	`jXX Dest`	
Memory		No operation

	`call Dest`	
Memory	M8[valE] $\leftarrow$ valP	Write return value on stack

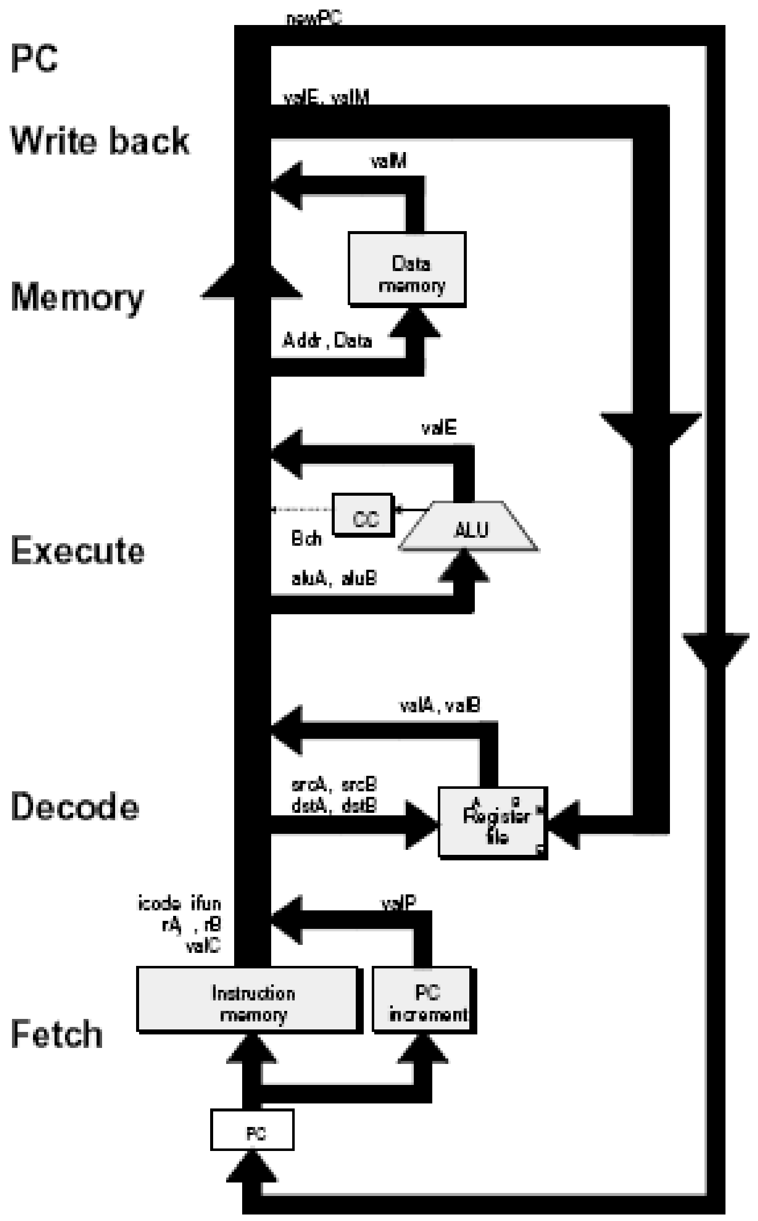
	`ret`	
Memory	valM $\leftarrow$ M8[valA]	Increment stack pointer

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

Write Back Logic

Notice for Write-back, there is no explicit hardware here.

That's because the location for writing back was determined at the decode stage. At this stage we have simply computed the values to write-back into the register file!



Destination E

Where to store the value computed by the ALU?

	`OPq rA,rB`	
Write-back	R[rB] $\leftarrow$ valE	Write back result

	`rmmovq rA,D(rB)`	
Write-back		None

	`popq rA`	
Write-back	R[`%rsp`] $\leftarrow$ valE	Update stack pointer

	`jXX Dest`	
Write-back		None

	`call Dest`	
Write-back	R[`%rsp`] $\leftarrow$ valE	Update stack pointer

	`ret`	
Write-back	R[`%rsp`] $\leftarrow$ valE	Update stack pointer

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

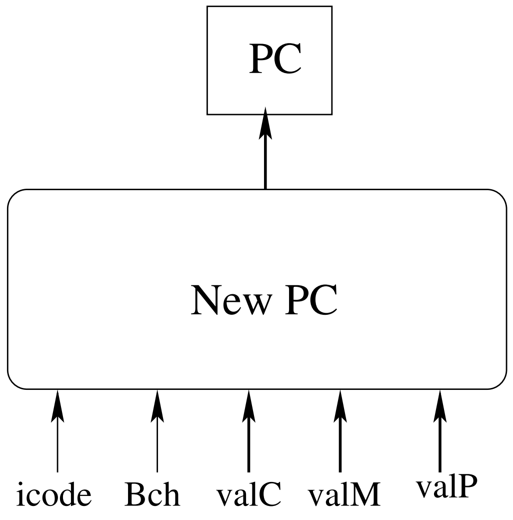
PC Update Logic

New PC

Select next value of PC.

Depends on:

- `icode`: current instruction
- `Bch`: result of branch logic
- `valC`: constant from instruction word
- `valM`: value from memory (stack)
- `valP`: predicted value from fetch



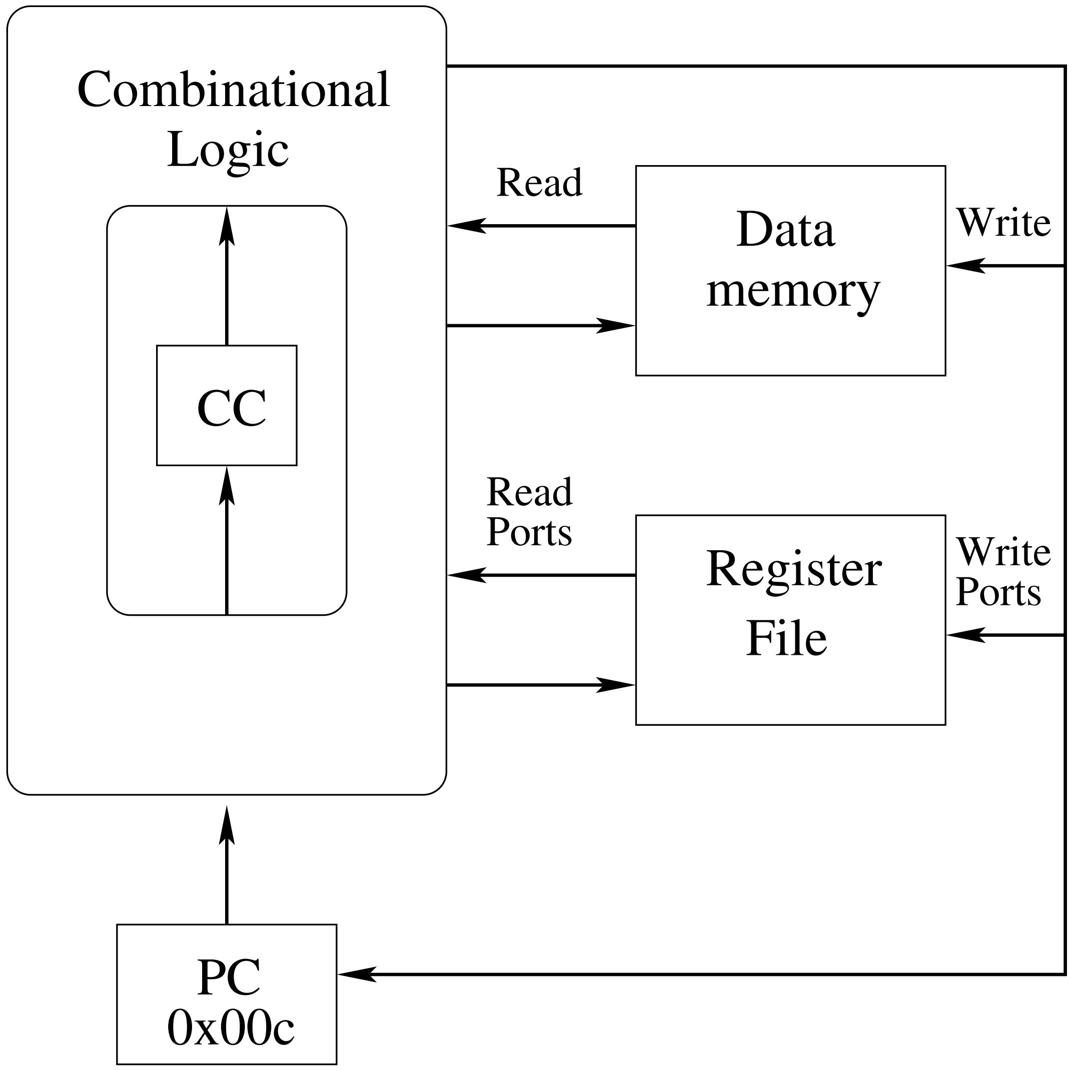
PC Update

What is the new value of the PC?

	`OPq rA,rB`	
PC update	PC ← valP	Update PC (by 2)
	`rmmovq rA,D(rB)`	
PC update	PC ← valP	Update PC (by 10)
	`popq rA`	
PC update	PC ← valP	Update PC (by 2)
	`jXX Dest`	
PC update	PC ← Bch ? valC : valP	Update PC (to what?)
	`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: All updated as the clock rises.

- PC register
- Condition Code register
- Register file

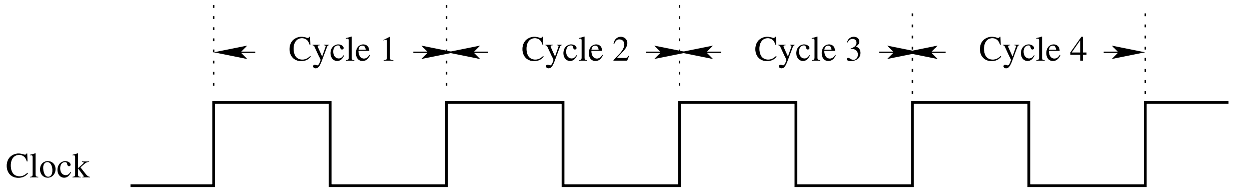
Combinational Logic

- ALU
- Control logic

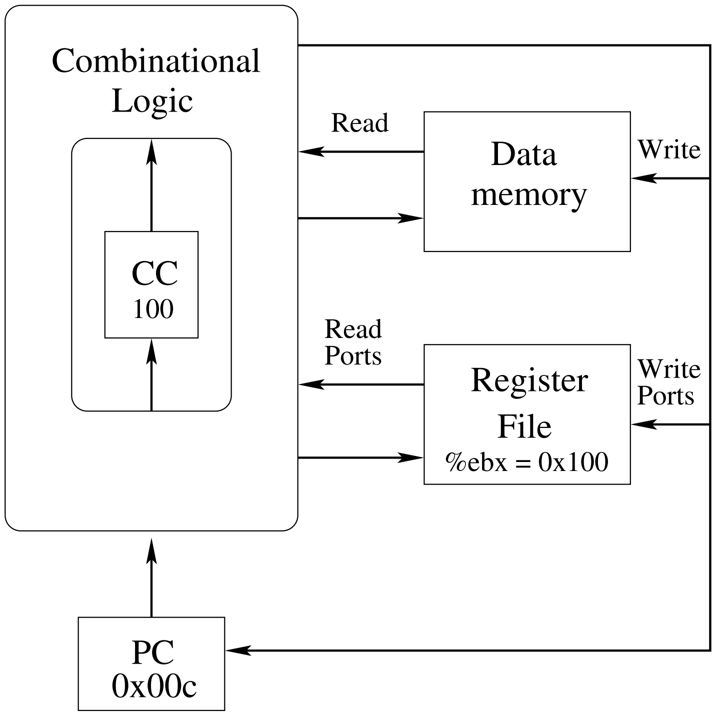
Sequential Logic

- Instruction memory
- Register file
- Data memory

SEQ Operation 2

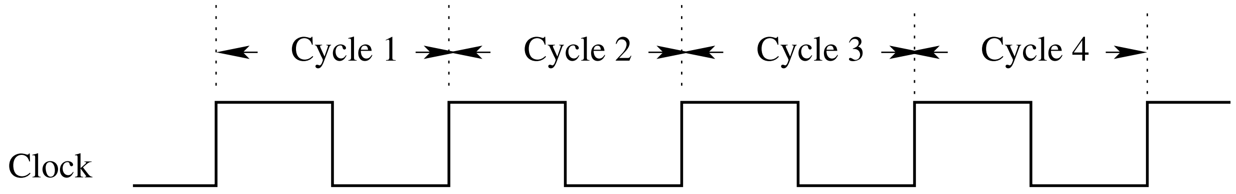


Cycle 1:	`0x000:    irmovq   $0x100,%rbx    # %rbx <-- 0x100`
Cycle 2:	`0x006:    irmovq   $0x200,%rdx    # %rdx <-- 0x200`
Cycle 3:	`0x00c:    addq     %rdx,%rbx      # %rbx <-- 0x300, CC <-- 000`
Cycle 4:	`0x00e:    je       dest           # Not taken`

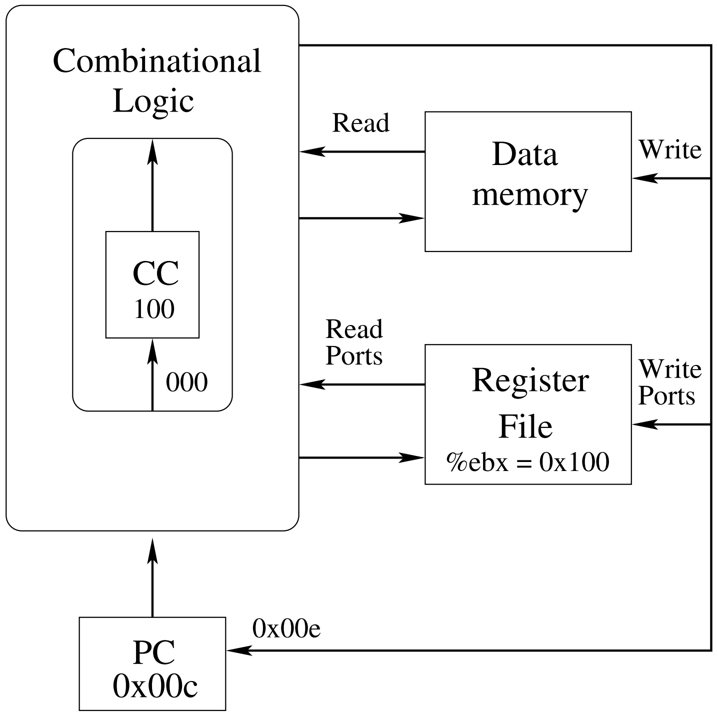


- state is set according to first irmovq instruction
- combinational logic is starting to react to state changes

SEQ Operation 3

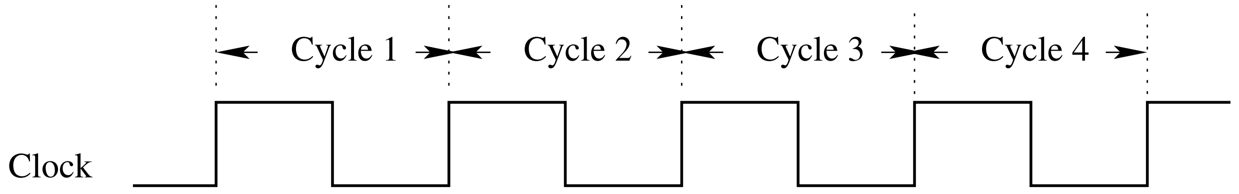


Cycle 1:	0x000:	irmovq	$0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x006:	irmovq	$0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x00c:	addq	%rdx,%rbx	# %rbx <-- 0x300, CC <-- 000
Cycle 4:	0x00e:	je	dest	# Not taken

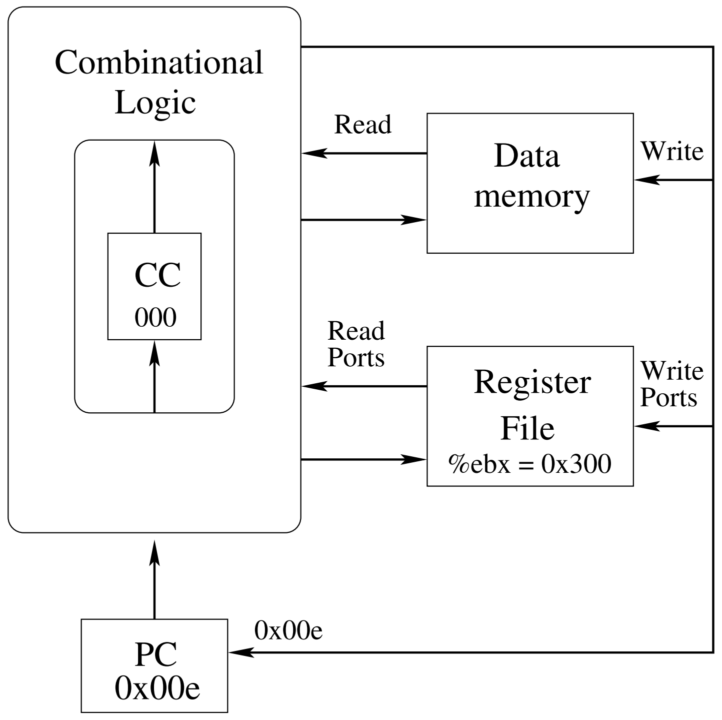


- state is set according to second irmovq instruction
- combinational logic generates results for addq instruction

SEQ Operation 4

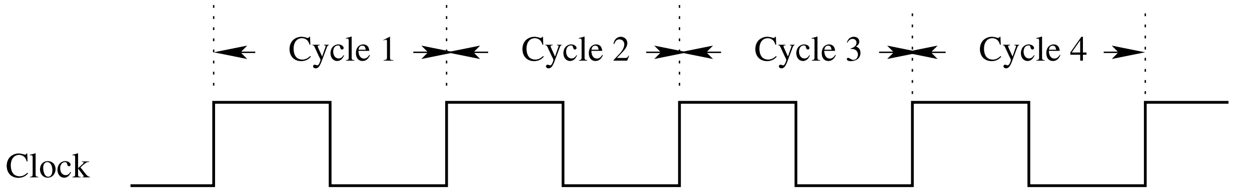


Cycle 1:	`0x000:`	`irmovq`	`$0x100,%rbx`	`# %rbx <-- 0x100`
Cycle 2:	`0x006:`	`irmovq`	`$0x200,%rdx`	`# %rdx <-- 0x200`
Cycle 3:	`0x00c:`	`addq`	`%rdx,%rbx`	`# %rbx <-- 0x300, CC <-- 000`
Cycle 4:	`0x00e:`	`je`	`dest`	`# Not taken`

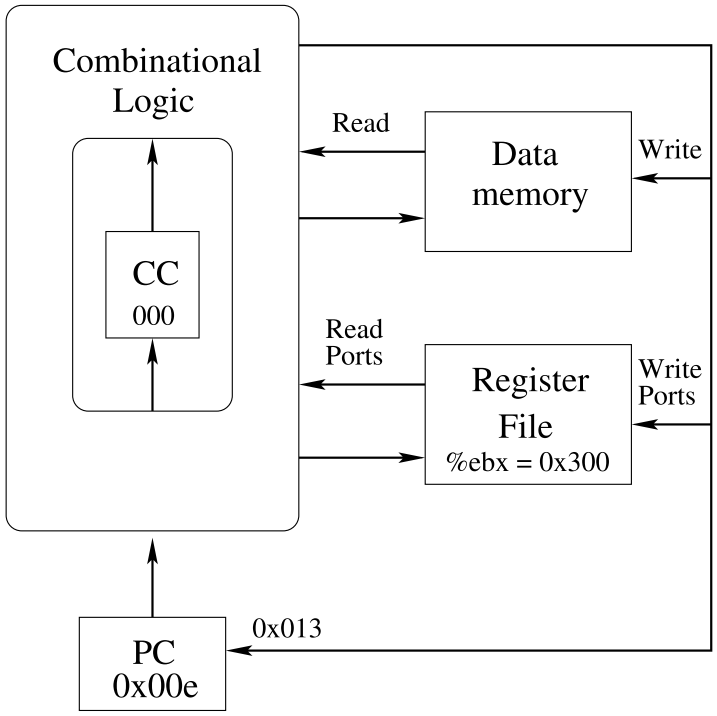


- state is set according to addq instruction
- combinational logic starting to react to state changes

SEQ Operation 5



Cycle 1:	0x000:	irmovq	$0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x006:	irmovq	$0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x00c:	addq	%rdx,%rbx	# %rbx <-- 0x300, CC <-- 000
Cycle 4:	0x00e:	je	dest	# Not taken



- state is set according to addq instruction
- combinational logic generates results for je instruction

SEQ Summary

Implementation

- Express every instruction as a 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. What is the slowest stage?
- In one cycle,must propagate through instruction memory, register file, ALU, and data memory.
- Would need to run the clock very slowly.
- Hardware units are only active for a fraction of the clock cycle.

Where We're Headed: Pipelining