CS429: Computer Organization and Architecture

Datapath I

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

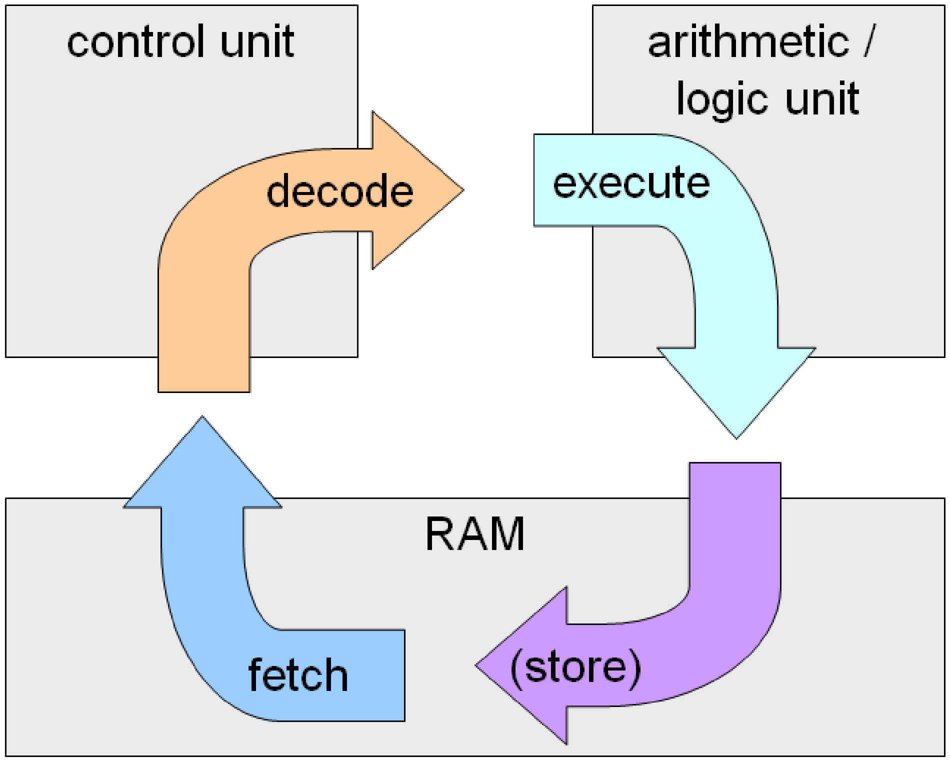
Last updated: March 25, 2019 at 13:29

Where We're Headed

Recall that the most fundamental abstraction for the machine semantics for the x86/Y86 or similar machines is the *fetch-decode-execute* cycle (the *von Neumann architecture*).

The machine repeats the following steps forever:

1. fetch the next instruction from memory (the PC tells you which is next);
2. decode the instruction (in the control unit);
3. execute the instruction;
4. update the state appropriately;
5. go to step 1.



Can We Speed Up the Fetch-Execute Cycle?

Notice that fetching, decoding, executing, updating likely *use different hardware modules*.

Imagine if we could do all of the following phases *in parallel*:

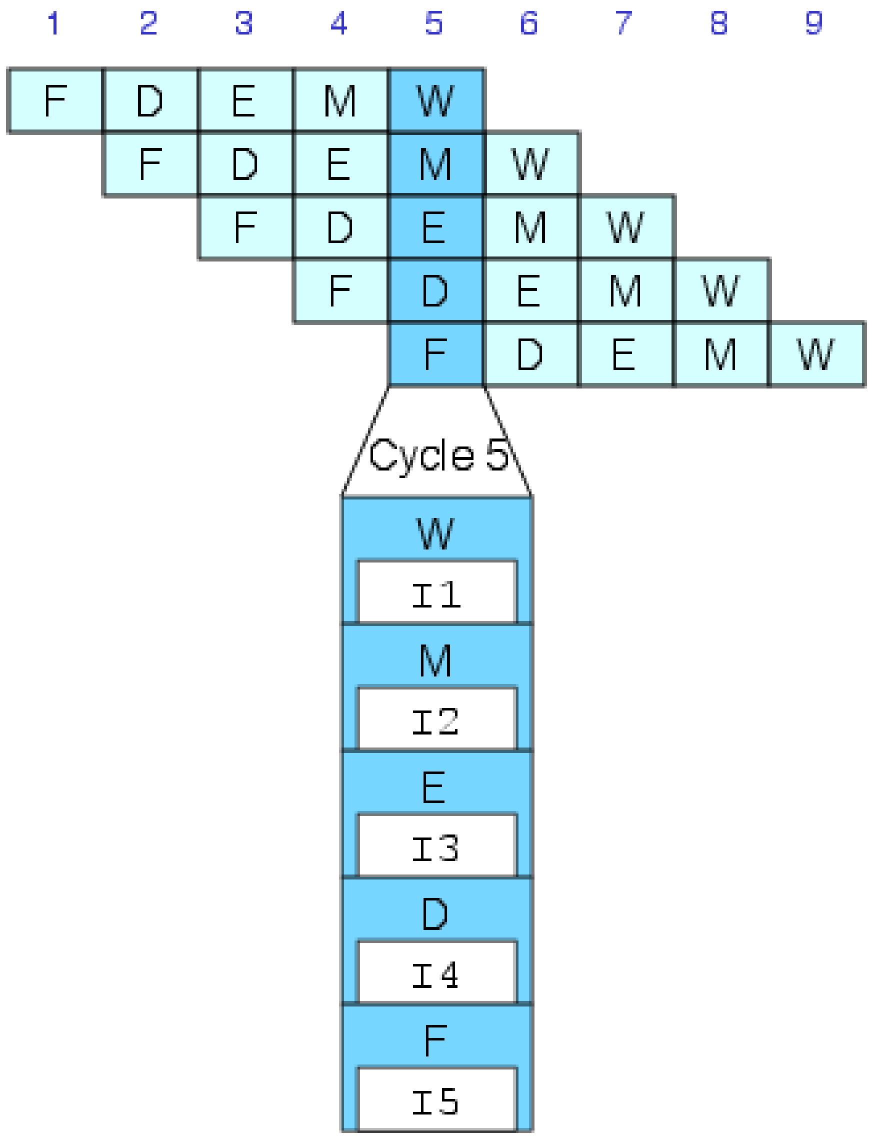
1. Update the state for instruction k,
2. Execute instruction $k + 1$
3. Decode instruction $k + 2$
4. Fetch instruction $k + 3$

This is called *pipelining*. It would allow much better utilization of the hardware and speed up the execution of our programs substantially!

We don't use exactly those phases, but something similar.

Looking Ahead: The Pipeline Ideal

```
irmovq    $1,%rax   #I1
irmovq    $2,%rbx   #I2
irmovq    $3,%rcx   #I3
irmovq    $4,%rdx   #I4
halt                #I5
```



Overview

How do we build a digital computer?

- Hardware building blocks: digital logic primitives.
- Instruction set architecture: what HW must implement.

Principled approach

- Hardware designed to implement one instruction at a time, and connect to the next instruction.
- Decompose each instruction into a series of steps.
- Expect that many steps will be common to many instructions.

Extend design from there

- Overlap execution of multiple instructions (pipelining).
- Parallel execution of many instructions.

Y86 Instruction Set

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								

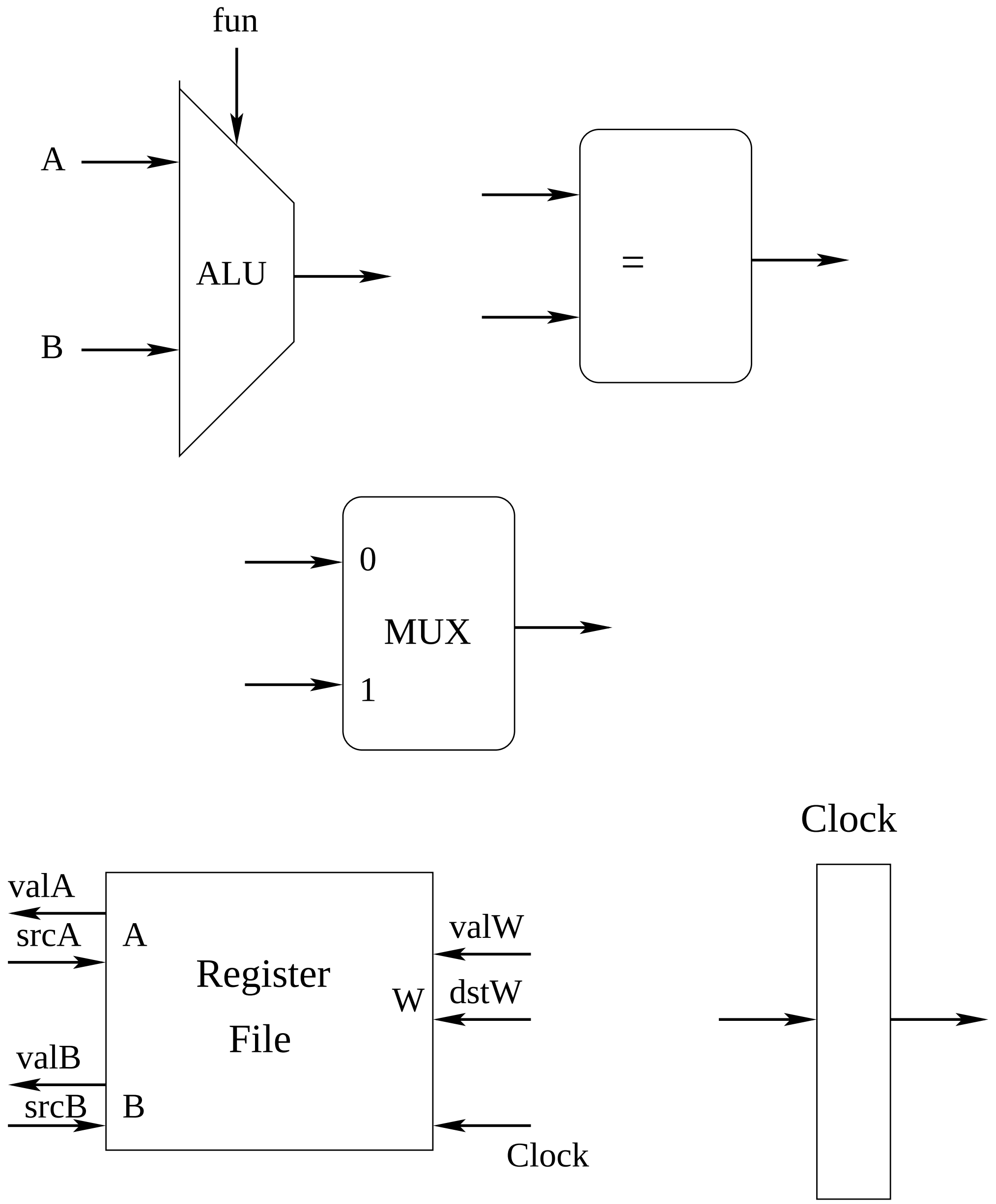
Building Blocks

Combinational Logic

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

Storage Elements

- Store bits
- Implement addressable memories
- Non-addressable registers
- Loaded only as clock rises.



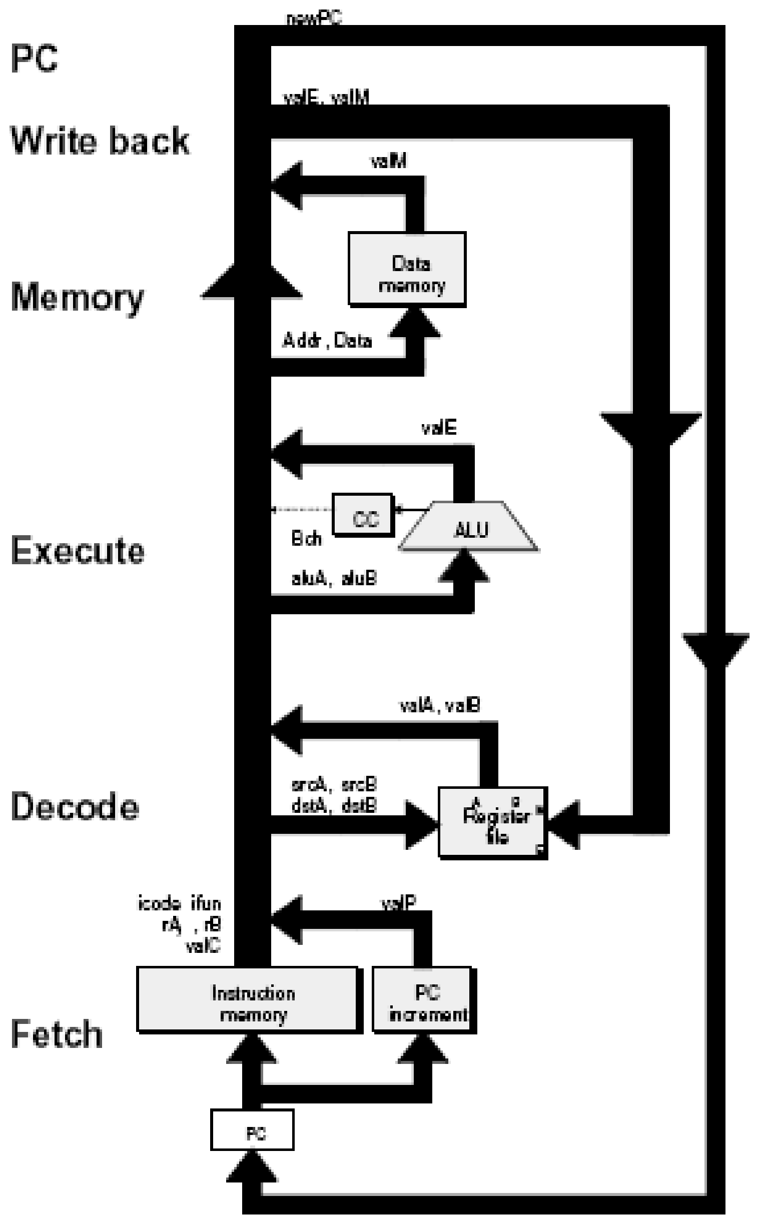
SEQ Hardware Structure

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



The Basic Idea

Break instruction execution into a series of common stages so that (eventually) multiple instructions can be processed concurrently.

Pros:

- Microcoding gives greater instruction granularity.
- May be able to use hardware more efficiently.
- Provides greater instruction throughput.

Challenges:

- Requires very careful ISA design.
- Assumes commonality among instruction types.
- Data and control hazards may inhibit pipelining.

SEQ Stages

Fetch: Read instruction from instruction memory.

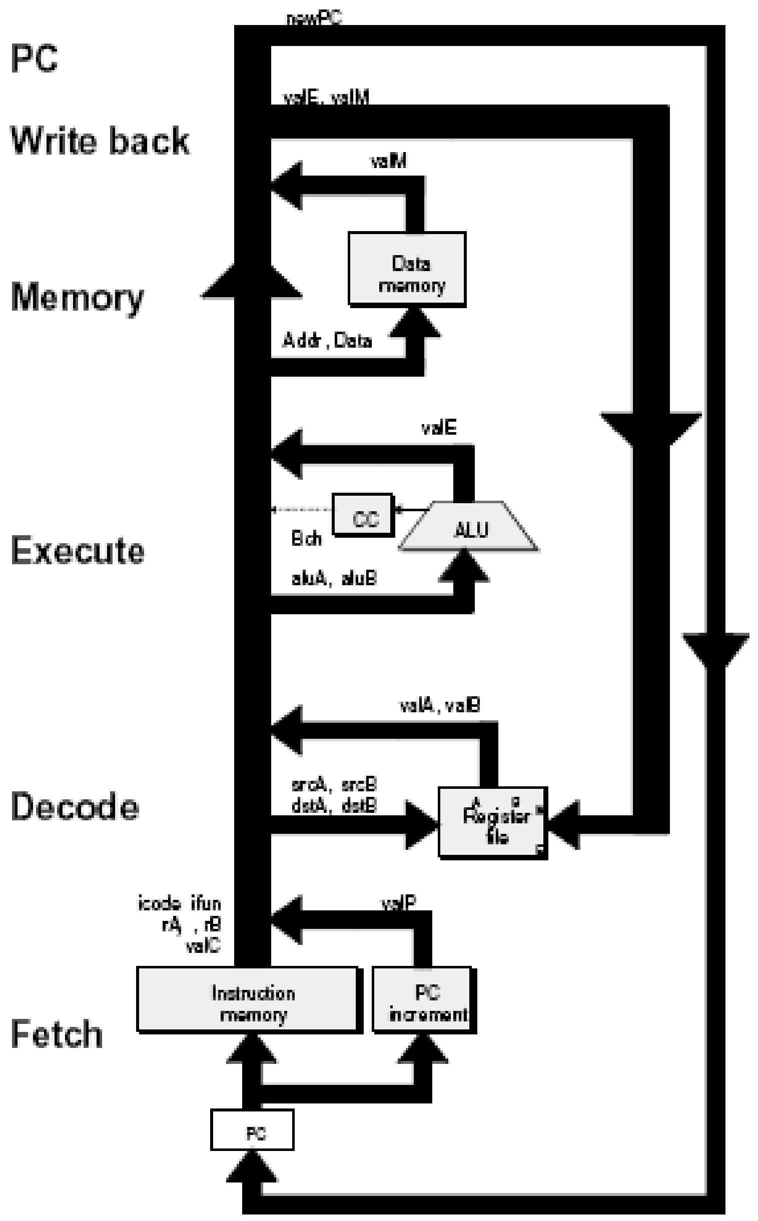
Decode: Read program registers

Execute: Compute value or address

Memory: Read or write back data.

Write Back: Write program registers.

PC: Update the program counter.



SEQ Stages

This is one possible decomposition of the instruction flow into stages. Each stage can be considered a “subroutine” in the Fetch / Decode / Execute cycle.

- **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.

Pipelining works best if *every* instruction can be decomposed into these same stages. Which do you think is probably the slowest stage?

Computed Values

Fetch

`icode`	Instruction code
`ifun`	Function code
`rA`	Inst. register A
`rB`	Inst. register B
`valC`	Instruction constant
`valP`	Incremented PC

Execute

`valE`	ALU result
`Bch`	Branch flag

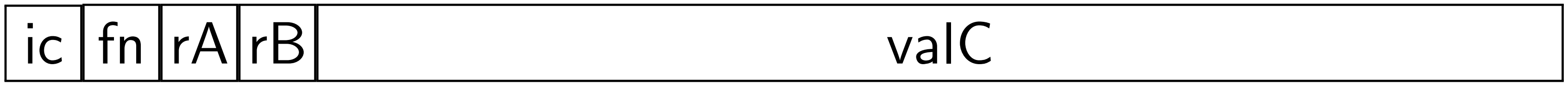
Decode

`srcA`	Register ID A
`srcB`	Register ID B
`dstE`	Dest. register E
`dstM`	Dest. register M
`valA`	Register value A
`valB`	Register value B

Memory

`valM`	Value from memory

Instruction Decoding



General Instruction Format

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

Executing Arith./Logical Operations

This is the general form of a Y86 arithmetic/logical operation.

```
OPq rA,rB
```

6	fn	rA	rB

What happens at each stage?

- **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.

Executing Arith./Logical Operations

`OPq rA,rB`

6	fn	rA	rB

Fetch: Read 2 bytes.*

Decode: Read operands (rA, rB).

Execute:

- Perform the operation with ALU.
- Set condition codes.

Memory: Do nothing.

Write back: Update dest. register (rB).

PC Update: Increment PC by 2. Why?

*The system probably reads 10+ bytes, not knowing in advance that this is a 2 byte instruction.

Stage Computation: Arith./Logical Ops

	`OPq rA,rB`	Comment
Fetch	icode:ifun $\leftarrow$ M1[PC] rA:rB $\leftarrow$ M1[PC+1] valP $\leftarrow$ PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[rB]	Read operand A Read operand B
Execute	valE $\leftarrow$ valB OP valA Set CC	Perform ALU operation Set condition code register
Memory		
Write back	R[rB] $\leftarrow$ valE	Write back result
PC Update	PC $\leftarrow$ valP	Update PC

- Formulate instruction execution as a sequence of simple steps.
- Use the same general form for all instructions.
- Why do this? Microcode?

Executing rmmovq

```
rmmovq rA,D(rB)
```

4	0	rA	rB	D

Fetch: Read 10 bytes.

Decode: Read operand regs.

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)`	Comment
Fetch	icode:ifun $\leftarrow$ M1[PC] rA:rB $\leftarrow$ M1[PC+1] valC $\leftarrow$ M8[PC+2] valP $\leftarrow$ PC+10	Read instruction byte Read register byte Read displacement D Compute next PC
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[rB]	Read operand A Read operand B
Execute	valE $\leftarrow$ valB + valC	Compute effective address
Memory	M8[valE] $\leftarrow$ valA	Write value to memory
Write back		
PC Update	PC $\leftarrow$ valP	Update PC

- Use the 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`	Comment
Fetch	icode:ifun $\leftarrow$ M1[PC]	Read instruction byte
	rA:rB $\leftarrow$ M1[PC+1]	Read register byte
	valP $\leftarrow$ PC+2	Compute next PC
Decode	valA $\leftarrow$ R[`%rsp`]	Read stack pointer
	valB $\leftarrow$ R[`%rsp`]	Read stack pointer
Execute	valE $\leftarrow$ valB + 8	Increment stack pointer
Memory	valM $\leftarrow$ M8[valA]	Read from stack.
Write back	R[`%rsp`] $\leftarrow$ valE	Update stack pointer
	R[rA] $\leftarrow$ valM	Write back result
PC Update	PC $\leftarrow$ valP	Update PC

- Use the ALU to increment stack pointer.
- Must update two registers: popped value, new stack pointer.

Executing Jumps

```
jXX Dest
```

7	fn	Dest

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 is taken.
- Otherwise, increment PC by 9.

Stage Computation: Jumps

	`jXX Dest`	Comment
Fetch	icode:ifun $\leftarrow$ M1[PC] valC $\leftarrow$ M8[PC+1] valP $\leftarrow$ PC+9	Read instruction byte Read destination address 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

`call Dest`	8	0	Dest

Fetch:

- Read 9 bytes
- Increment PC by 9

Decode: Read stack pointer.

Execute: Decrement stack pointer by 8.

Memory:

- Write incremented PC (return address) to new value of stack pointer.

Write back: Update stack pointer.

PC Update: Set PC to Dest

Stage Computation: call

	`call Dest`	Comment
Fetch	icode:ifun $\leftarrow$ M1[PC] valC $\leftarrow$ M8[PC+1] valP $\leftarrow$ PC+9	Read instruction byte Read destination address Compute return addr.
Decode	valB $\leftarrow$ R[`%rsp`]	Read stack pointer
Execute	valE $\leftarrow$ valB + -8	Decrement stack pointer
Memory	M8[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 the ALU to decrement stack pointer.
- Store incremented PC.

Executing ret

`ret` | 9 | 0 |
|---|---|

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`	Comment
Fetch	icode:ifun $\leftarrow$ M1[PC]	Read instruction byte
Decode	valA $\leftarrow$ R[`%rsp`]	Read operand stack
	valB $\leftarrow$ R[`%rsp`]	Read operand stack
Execute	valE $\leftarrow$ valB $+$ 8	Increment stack pointer
Memory	valM $\leftarrow$ M8[valA]	Read return address
Write back	R[`%rsp`] $\leftarrow$ valE	Update stack pointer
PC Update	PC $\leftarrow$ valM	Set PC to return address

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

Computation Steps: ALU Operations

		`OPq rA,rB`	Comment
Fetch	icode,ifun	icode:ifun ← M1[PC]	Read instruction byte
	rA,rB	ra:rB ← M1[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,srcA	valB ← R[rB]	Read operand B
Execute	valE	valE ← valB OP valA	Perform ALU operation
	Cond code	Set CC	Set condition code reg.
Memory	valM		Memory read/write
Write back	dstE	R[rB] ← valE	Write back ALU result
	dstM		Write back memory
PC update	PC	PC ← valP	Update PC

- All instructions follow the same general pattern.
- They differ only in what gets computed each step.

Computation Steps: Call

		`call Dest`	Comment
Fetch	icode,ifun	icode:ifun $\leftarrow$ M1[PC]	Read instruction byte
	rA,rB		Read register byte
	valC	valC $\leftarrow$ M8[PC+1]	Read constant word
	valP	valP $\leftarrow$ PC+9	Compute next PC
Decode	valA,srcA		Read operand A
	valB,srcA	valB $\leftarrow$ R[`%rsp`]	Read operand B
Execute	valE	valE $\leftarrow$ valB - 8	Perform ALU operation
	Cond code		Set condition code reg.
Memory	valM	M8[valE] $\leftarrow$ valP	Memory read/write
Write back	dstE	R[`%rsp`] $\leftarrow$ valE	Write back ALU result
	dstM		Write back memory
PC update	PC	PC $\leftarrow$ valC	Update PC

- All instructions follow the same general pattern.
- They differ only in what gets computed each step.

Computed Values

Fetch

`icode`	Instruction code
`ifun`	Function code
`rA`	Inst. register A
`rB`	Inst. register B
`valC`	Instruction constant
`valP`	Incremented PC

Execute

`valE`	ALU result
`Bch`	Branch flag

Decode

`srcA`	Register ID A
`srcB`	Register ID B
`dstE`	Dest. register E
`dstM`	Dest. register M
`valA`	Register value A
`valB`	Register value B

Memory

`valM`	Value from memory

Summary

- Sequential instruction execution cycle.
- Instruction mapping to hardware.
- Instruction decoding.