CS429: Computer Organization and Architecture

Datapath I

Warren Hunt, Jr. and Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: November 12, 2014 at 07:34

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). More on that later.
- Parallel execution of many instructions.
- Covered in more advanced computer architecture course.

Y86 Instruction Set

Byte	0	1	2	3	4	5
`halt`	0	0				
`nop`	1	0				
`rrmovl rA,rB`	2	0	rA	rB		
`irmovl V,rB`	3	0	F	rB	V	
`rmmovl rA,D(rB)`	4	0	rA	rB	D	
`mrmovl D(rB),rA`	5	0	rA	rB	D	
`OP1 rA,rB`	6	fn	rA	rB		
`jXX Dest`	7	fn	Dest			
`call Dest`	8	0	Dest			
`ret`	9	0				
`pushl rA`	A	0	rA	F		
`popl rA`	B	0	rA	F		

Y86 Function Codes

These are the function codes for specific instances of the OP1 and jXX instructions.

`addl`	6	0
`subl`	6	1
`andl`	6	2
`xorl`	6	3

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

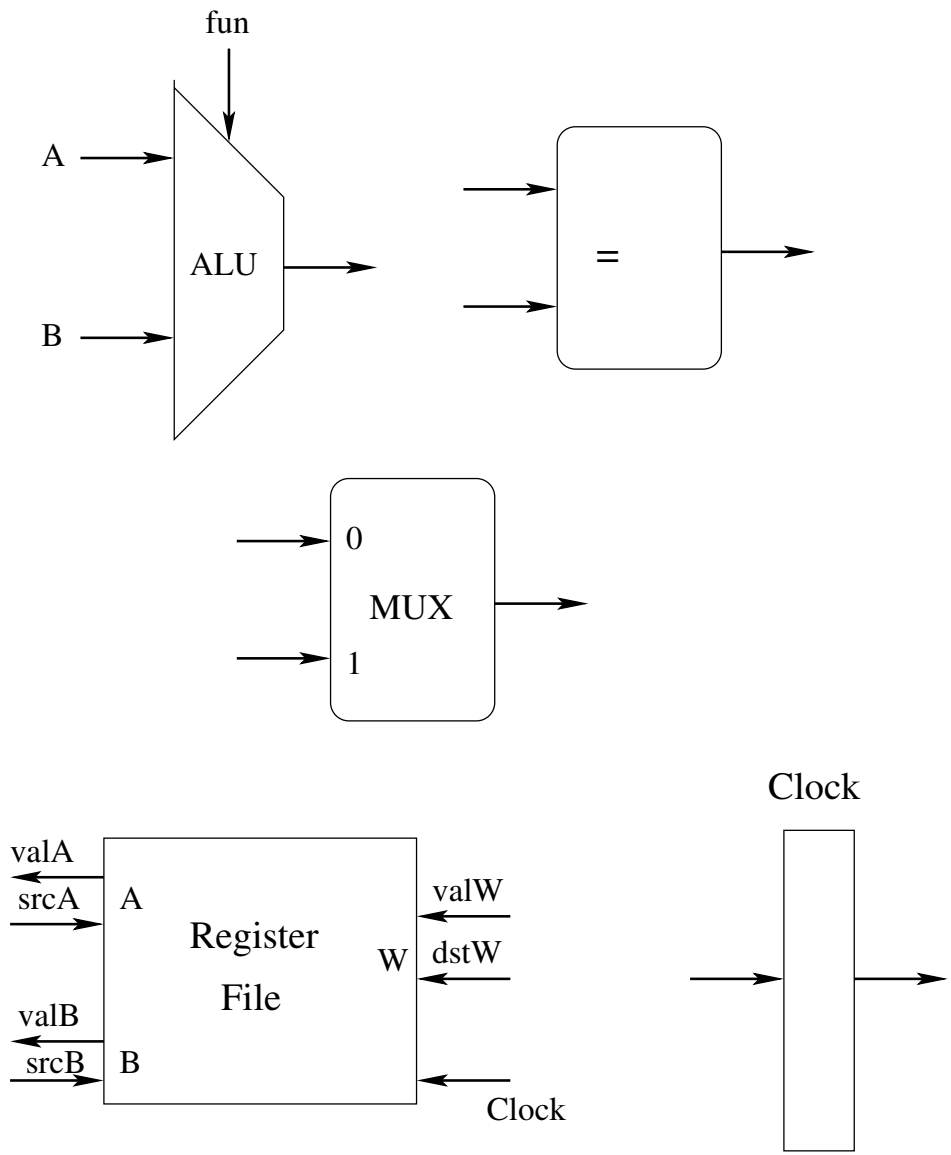
Building Blocks

Combinational Logic

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

Storage Elements

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



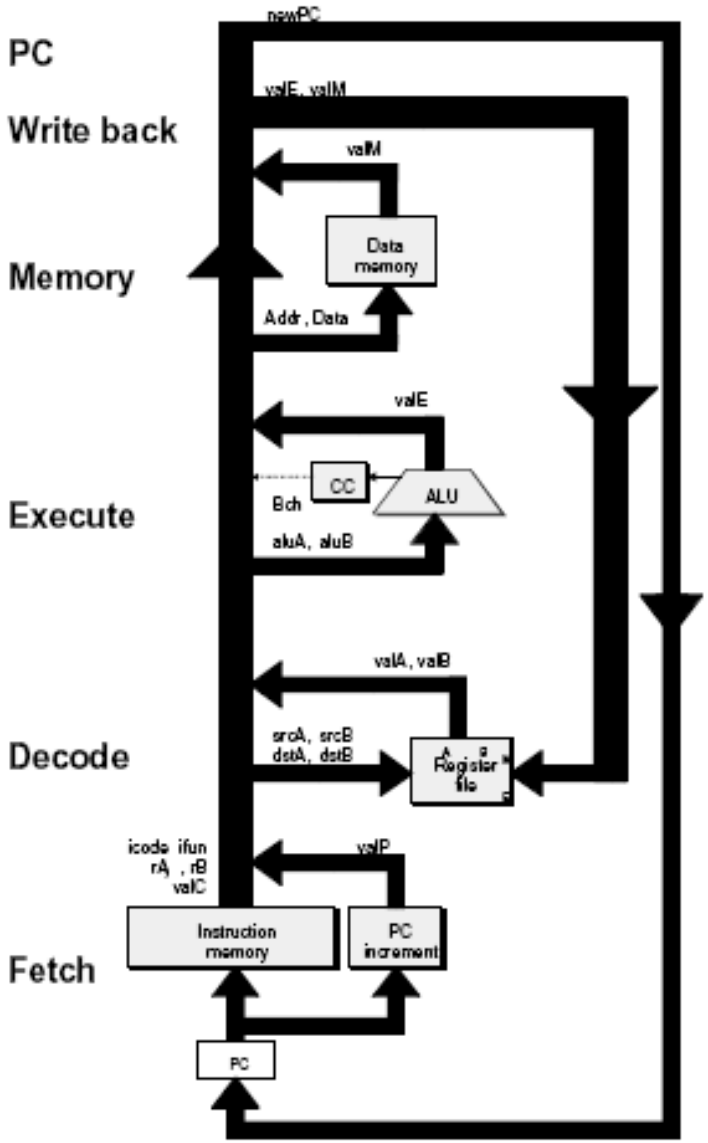
SEQ Hardware Structure

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



SEQ Stages

Fetch: Read instruction from instruction memory.

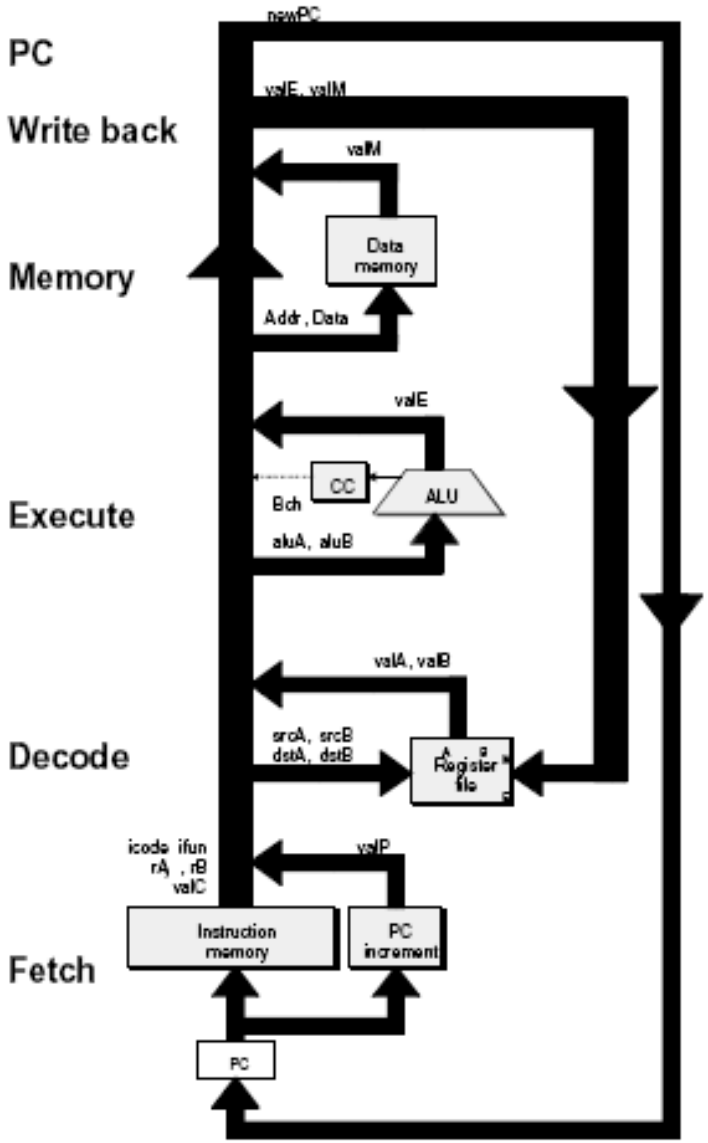
Decode: Read program registers

Execute: Compute value or address

Memory: Read or write back data.

Write Back: Write program registers.

PC: Update the program counter.



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

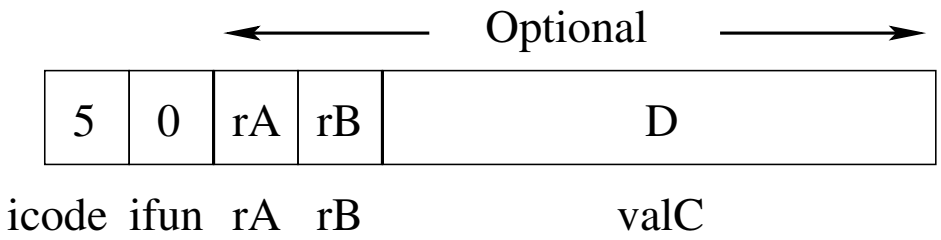
Decode

`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

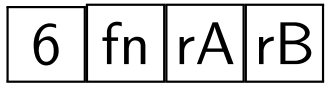


Instruction Format

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

Executing Arith./Logical Operations

`OP1 rA,rB`



Fetch: Read 2 bytes.

Decode: Read operand regs.

Execute:

- Perform the operation.
- Set condition codes.

Memory: Do nothing.

Write back: Update register.

PC Update:

- Increment PC by 2.
- Why?

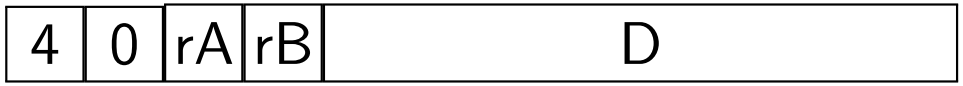
Stage Computation: Arith./Logical Ops

	`OP1 rA,rB`	Comment
Fetch	icode:ifun ← M1[PC] rA:rB ← M1[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 a sequence of simple steps.
- Use the same general form for all instructions.
- Why do this? Microcode?

Executing rmmovl

`rmmovl rA,D(rB)`



Fetch: Read 6 bytes.

Decode: Read operand regs.

Execute: Compute effective address.

Memory: Write to memory.

Write back: Do nothing.

PC Update: Increment PC by 6.

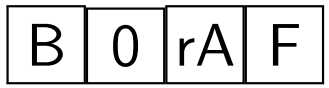
Stage Computation: rmmovl

	`rmmovl rA, D(rB)`	Comment
Fetch	icode:ifun ← M1[PC]	Read instruction byte
	rA:rB ← M1[PC+1]	Read register byte
	valC ← M4[PC+2]	Read displacement D
	valP ← PC+6	Compute next PC
Decode	valA ← R[rA]	Read operand A
	valB ← R[rB]	Read operand B
Execute	valE ← valB + valC	Compute effective address
Memory	M4[valE] ← valA	Write value to memory
Write back		
PC Update	PC ← valP	Update PC

- Use the ALU for address computation.

Executing popl

`popl rA`



Fetch: Read 2 bytes.

Decode: Read stack pointer.

Execute: Increment stack pointer by 4.

Memory: Read from old stack pointer.

Write back:

- Update stack pointer.
- Write result to register.

PC Update: Increment PC by 2.

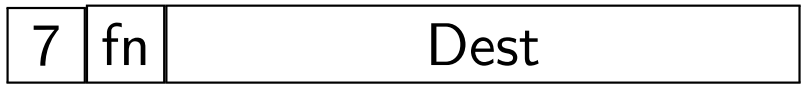
Stage Computation: popl

	`popl rA`	Comment
Fetch	icode:ifun ← M1[PC] rA:rB ← M1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA ← R[%esp] valB ← R[%esp]	Read stack pointer Read stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M4[valA]	Read from stack.
Write back	R[%esp] ← valE R[rA] ← valM	Update stack pointer Write back result
PC Update	PC ← valP	Update PC

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

Executing Jumps

`jXX Dest`



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 is taken.
- Otherwise, increment PC.

Stage Computation: Jumps

	`jXX Dest`	Comment
Fetch	icode:ifun ← M1[PC] valC ← M4[PC+1] valP ← PC+5	Read instruction byte Read destination address Fall through address
Decode		
Execute	Bch ← Cond(CC, ifun)	Take branch?
Memory		
Write back		
PC Update	PC ← Bch ? valC : valP	Update PC

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

Executing call

`call Dest`

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.

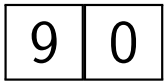
PC Update: Set PC to Dest

Stage Computation: call

	`call Dest`	Comment
Fetch	icode:ifun ← M1[PC] valC ← M4[PC+1] valP ← PC+5	Read instruction byte Read destination address Compute return point
Decode	valB ← R[%esp]	Read stack pointer
Execute	valE ← valB + -4	Decrement stack pointer
Memory	M4[valE] ← valP	Write return value on stack.
Write back	R[%esp] ← valP	Update stack pointer
PC Update	PC ← valC	Set PC to destination.

- Use the ALU to decrement stack pointer.
- Store incremented PC.

Executing ret

`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`	Comment
Fetch	icode:ifun ← M1[PC]	Read instruction byte
Decode	valA ← R[%esp] valB ← R[%esp]	Read operand stack Read operand stack
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M4[valA]	Read return address
Write back	R[%esp] ← valE	Update stack pointer
PC Update	PC ← valM	Set PC to return address

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

Computation Steps: ALU Operations

		`OP1 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 ← M1[PC]	Read instruction byte
	rA,rB		Read register byte
	valC	valC ← M4[PC+1]	Read constant word
	valP	valP ← PC+5	Compute next PC
Decode	valA,srcA		Read operand A
	valB,srcA	valB ← R[%esp]	Read operand B
Execute	valE	valE ← valB - 4	Perform ALU operation
	Cond code		Set condition code reg.
Memory	valM	M4[valE] ← valP	Memory read/write
Write back	dstE	R[%esp] ← valE	Write back ALU result
	dstM		Write back memory
PC update	PC	PC ← 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.