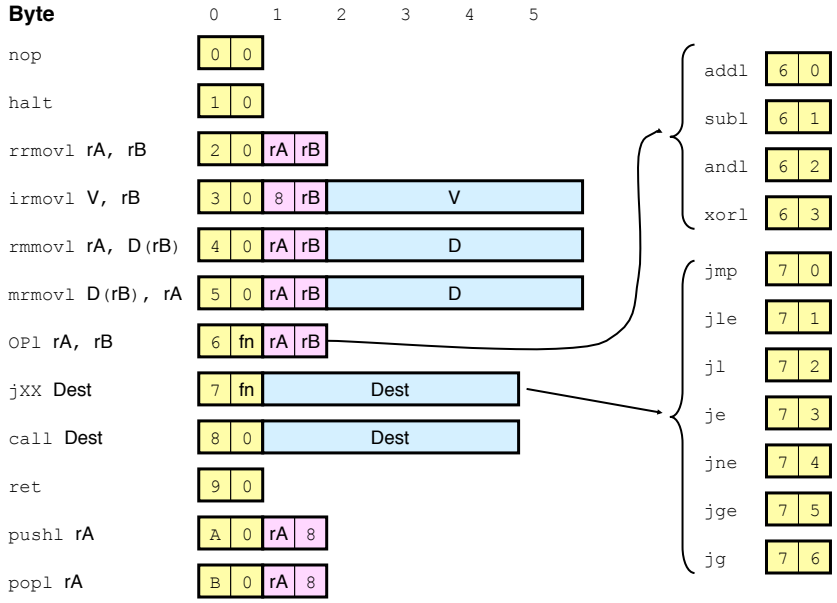
CS 270: Computer Organization

Sequential Implementation

Instructor:

Professor Stephen P. Carl

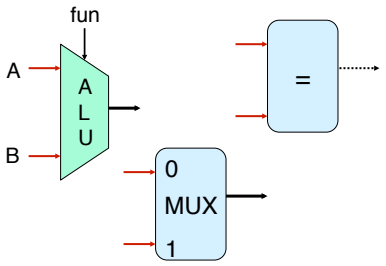
Y86 Instruction Set



Building Blocks

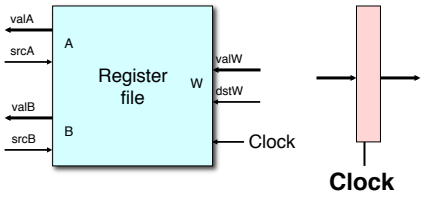
Combinational Logic

- 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

Key:

Predesigned hardware blocks in blue

- E.g., memories, ALU

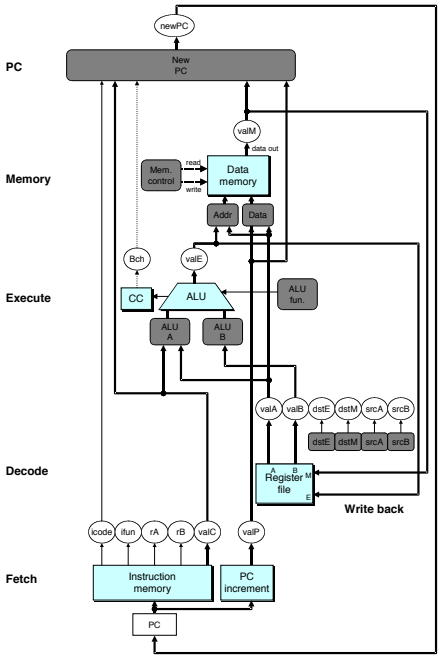
Control logic in gray

- Described in HCL

Thick lines: 32-bit word values

Thin lines: 4-8 bit values

Dotted lines: 1-bit values



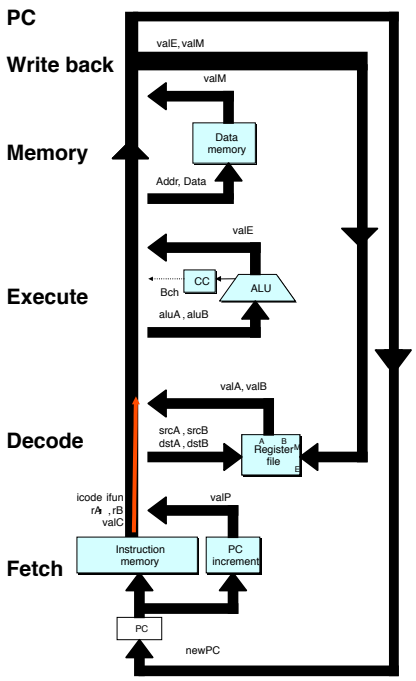
SEQ Hardware Structure

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



Sewanee: The University of the South

SEQ Hardware Structure

Fetch
- Read instruction from instruction memory

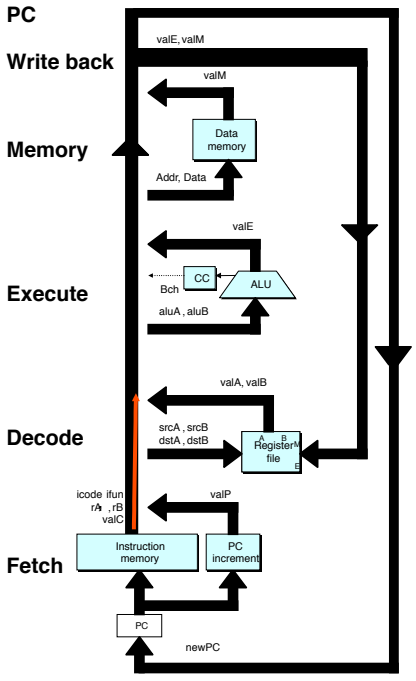
Decode
- Read program registers

Execute
- Compute value or address

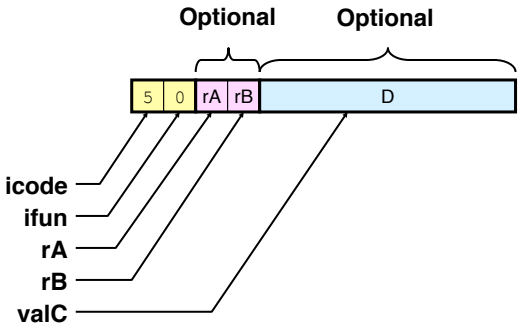
Memory
- Read or write data

Write Back
- Write program registers

PC
- Update program counter



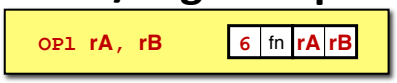
Instruction Decoding



Instruction Format

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

Executing Arith./Logical Operation



Fetch

- Read 2 bytes

Decode

- Read operand registers

Execute

- Perform operation
- Set condition codes

Memory

- Do nothing

Write back

- Update register

PC Update

- Increment PC by 2

Stage Computation: Arith/Log. Ops

	OPl rA, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[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

- Here we formulate instruction execution as sequence of small steps
- Use the same general form for all instructions

Executing `rmmovl`

Fetch
- Read 6 bytes

Decode
- Read operand registers

Execute
- Compute effective address

Memory
- Write to memory

Write back
- Do nothing

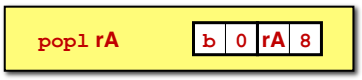
PC Update
- Increment PC by 6

Stage Computation: `rmmovl`

	`rmmovl` rA, D(rB)	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_4[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	M_4[valE] ← valA	Write value to memory
Write back		
PC update	PC ← valP	Update PC

Note use of ALU for address computation

Executing `popl`



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

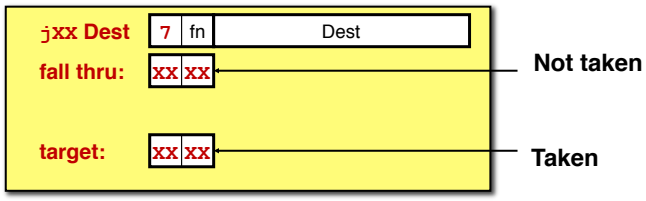
PC Update
- Increment PC by 2

Stage Computation: `popl`

	`popl` rA	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[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 ← M_4[valA]	Read from stack
Write back	R[`%esp`] ← valE R[rA] ← valM	Update stack pointer Write back result
PC update	PC ← valP	Update PC

- Note use of ALU to increment stack pointer
- Must update two registers
  - Popped value
  - New stack pointer

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

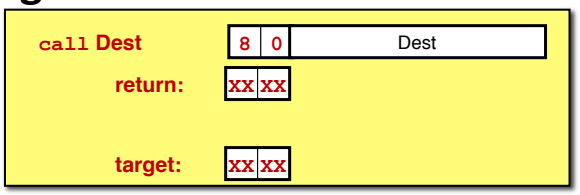
PC Update
- Set PC to Dest if branch taken or to incremented PC if not branch

Stage Computation: Jumps

	jXX Dest	
Fetch	icode:ifun ← M_1[PC] valC ← M_4[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`



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

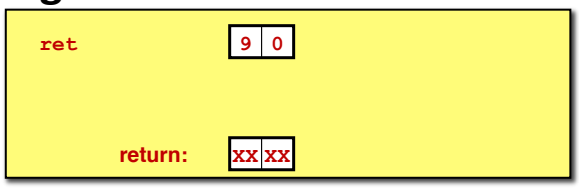
PC Update
- Set PC to Dest

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 ← M_1[PC]	Read instruction byte
	rA,rB		[Read register byte]
	valC	valC ← M_4[PC+1]	Read constant word
	valP	valP ← PC+5	Compute next PC
Decode	valA, srcA		[Read operand A]
	valB, srcB	valB ← R[`%esp`]	Read operand B
Execute	valE	valE ← valB + –4	Perform ALU operation
	Cond code		[Set condition code reg.]
Memory	valM	M_4[valE] ← valP	[Memory read/write]
Write back	dstE	R[`%esp`] ← valE	[Write back ALU result]
	dstM		Write back memory result
PC update	PC	PC ← 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:

Predesigned hardware blocks in blue

- E.g., memories, ALU

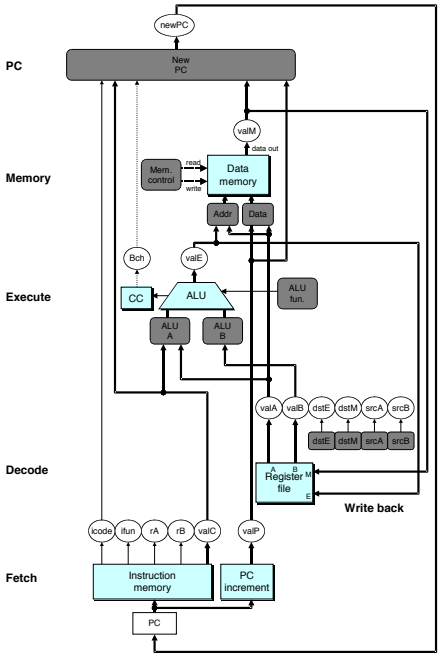
Control logic in gray

- Described in HCL

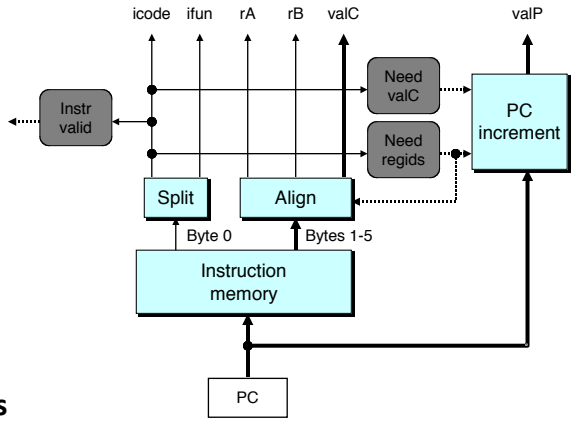
Thick lines: 32-bit word values

Thin lines: 4-8 bit values

Dotted lines: 1-bit values



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

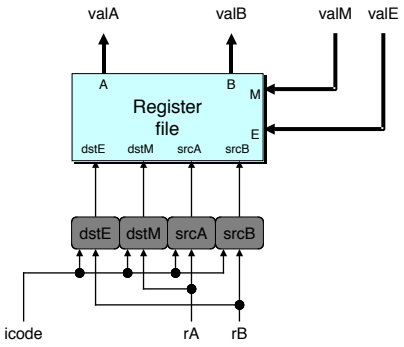
Decode Logic

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



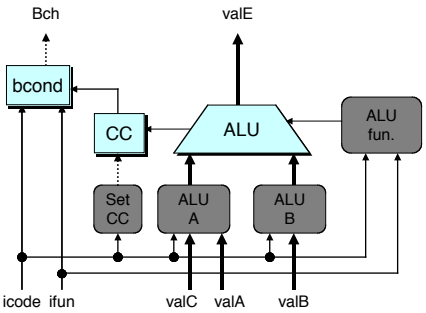
Execute Logic

Units

- ALU
  - Implements 4 A/L 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?



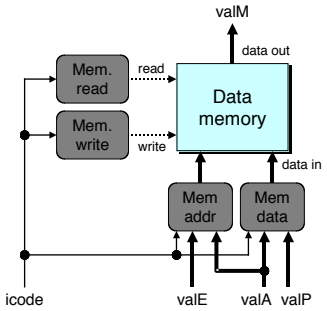
Memory Logic

Memory

- Reads or writes memory word

Control Logic

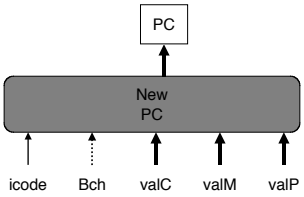
- Mem. read: should word be read?
- Mem. write: should word be written?
- Mem. addr.: Select address
- Mem. data.: Select data



PC Update Logic

New PC

- Select next value of PC



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