Sequential Implementation

CSC 235 - Computer Organization

References

- Slides adapted from CMU

Y86-64 Instruction Set

Byte	0	1	2	3	4	5	6	7	8	9

`halt`

`0` `0`

`nop`

`1` `0`

`rrmovq rA, rB`

`2` `0` `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

`cmovXX rA, rB`

`2` `fn` `rA` `rB`

`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
    - Addressable memories
    - Non-addressable registers
    - Loaded only as clock rises

Sequential Hardware Structure

- State
    - Program counter register (PC)
    - Condition code register (CC)
    - Register file
    - Memories
- Instruction flow
    - Read instruction at address specified by PC
    - Process through stages
    - Update program counter

Sequential Stages

- Fetch: read instruction from memory
- Decode: read program registers
- Execute: compute value or address
- Memory: read or write data
- Write back: write program registers
- PC: update program counter

Sequential Hardware Structure
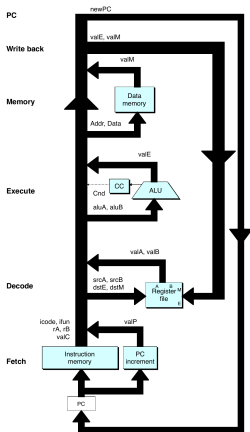
Instruction Decoding

- Instruction format (10 bytes max)
    - Instruction byte: `icode:ifun`
    - Optional register byte: `rA:rB`
    - Optional constant word: `valC`

Executing Arithmetic/Logical Operation

- Fetch: read 2 bytes
- Decode: read operand registers
- Execute: perform operation and set condition codes
- Memory: do nothing
- Write back: update register
- PC update: increment PC by 2

Stage Computation: Arithmetic/Logical Operations

Stage	Op rA rB	Action
Fetch	`icode:ifun = M1[PC]`	read instruction byte
	`rA:rB = M1[PC+1]`	read register byte
	`valP = PC+2`	compute next PC
Decode	`valA = R[rA]`	read operand A
	`valB = R[rB]`	read operand B
Execute	`valE = valB OP valA`	Perform ALU operation
Memory		
Write back	`R[rB] = valE`	Write back result
PC update	`PC = valP`	update PC

Executing `rmmovq`

- Fetch: read 10 bytes
- Decode: read operand registers
- Execute: compute effective address
- Memory: write to memory
- Write back: do nothing
- PC update: increment PC by 10

Stage Computation: `rmmovq`

Stage	rmmovq rA, D(rB)	Action
Fetch	`icode:ifun = M1[PC]`	read instruction byte
	`rA:rB = M1[PC+1]`	read register byte
	`valC = M8[PC+2]`	read 8 byte displacement
	`valP = PC+10`	compute next PC
Decode	`valA = R[rA]`	read operand A
	`valB = R[rB]`	read operand B
Execute	`valE = valB + valC`	compute effective address (ALU)
Memory	`M8[valE] = valA`	write 8 byte value to memory
Write back		
PC update	`PC = valP`	update PC

Executing `popq`

- 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 and write result to register
- PC update: increment PC by 2

Stage Computation: popq

Stage	popq rA	Action
Fetch	icode:ifun = M1[PC]	read instruction byte
	rA:rB = M1[PC+1]	read register byte
	valP = PC+2	compute next PC
Decode	valA = R[%rsp]	read stack pointer
	valB = R[%rsp]	read stack pointer
Execute	valE = valB + 8	increment stack pointer (ALU)
Memory	valM = M8[valA]	read 8 bytes from stack
Write back	R[%rsp] = valE	update stack pointer
	R[rA] = valM	write back result
PC update	PC = valP	update PC

Executing Conditional Moves

- Fetch: read 2 bytes
- Decode: read operand registers
- Execute: if not condition, then set destination register to 0xF
- Memory: do nothing
- Write back: update register (or not)
- PC update: increment PC by 2

Stage Computation: Conditional Move

Stage	cmovXX rA, rB	Action
Fetch	icode:ifun = M1[PC]	read instruction byte
	rA:rB = M1[PC+1]	read register byte
	valP = PC+2	compute next PC
Decode	valA = R[rA]	read operand A
	valB = 0	read stack pointer
Execute	valE = valB + valA	pass val through ALU (v
	if !Cond(CC,ifun) rB = 0xF	(disable register update)
Memory		
Write back	R[rB] = valE	write back result
PC update	PC = valP	update PC

Executing Jumps

- Fetch: read 9 bytes and increment PC by 9
- Decode: do nothing
- Execute: determine whether to take branch based on jump condition codes
- Memory: do nothing
- Write back: do nothing
- PC update: set PC to destination if branch taken or to incremented PC if not branch

Stage Computation: Jumps

Stage	jXX Dest	Action
Fetch	icode:ifun = M1[PC]	read instruction byte
	valC = M8[PC+1]	read 8 byte destination address
	valP = PC+9	fall through address
Decode		
Execute	Cnd = Cond(CC, ifun)	take branch?
Memory		
Write back		
PC update	PC = Cnd ? valC : valP	update PC

Executing `call`

- Fetch: read 9 bytes and increment PC by 9
- Decode: read stack pointer
- Execute: decrement stack pointer by 8
- Memory: write incremented PC to new value of stack pointer
- Write back: update stack pointer
- PC update: set PC to Dest

Stage Computation: `call`

Stage	call Dest	Action
Fetch	icode:ifun = M1[PC]	read instruction byte
	valC = M8[PC+1]	read 8 byte destination address
	valP = PC+9	compute return point
Decode	valB = R[%rsp]	read stack pointer
Execute	valE = valB + -8	decrement stack pointer (ALU)
Memory	M8[valE] = valP w	rite 8 byte return value on stack
Write back	R[%rsp] = valE	update stack pointer
PC update	PC = valC	set PC to destination

Executing ret

- 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`

Stage	`ret`	Action
Fetch	icode:ifun = M1[PC]	read instruction byte
Decode	valA = R[%rsp]	read operand stack pointer
	valB = R[%rsp]	read operand stack pointer
Execute	valE = valB + 8	increment stack pointer (ALU)
Memory	valM = M8[valA]	read return address
Write back	R[%rsp] = valE	update stack pointer
PC update	PC = valM	set PC to return address

Computation Steps

Stage	Steps	Action
Fetch	`icode:ifun`	read instruction byte
	`rA, rB`	[read register byte]
	`valC`	[read constant word]
	`valP`	compute next PC
Decode	`valA, srcA`	[read operand A]
	`valB, srcB`	[read operand B]
Execute	`valE`	perform ALU operation
	`Cond code`	[set/use condition code]
Memory	`valM`	[memory read/write]
Write back	`dstE`	[write back ALU result]
	`dstM`	[write back memory result]
PC update	`PC`	update PC

Computed Values

- Fetch
  - `icode`: instruction code
  - `ifun`: instruction function
  - `rA`: instruction register A
  - `rB`: instruction register B
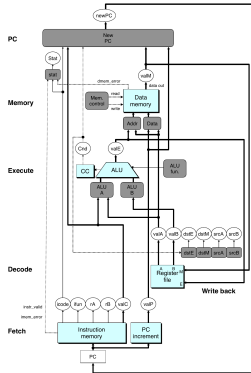 - `valC`: instruction constant
  - `valP`: incremented PC

Computed Values

- 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

Computed Values

- Execute
    - `valE`: ALU result
    - `Cnd`: branch/move flag
- Memory
    - `valM`: value from memory

Sequential Hardware

Sequential Hardware

- Diagram key
    - Blue boxes: pre-designed hardware blocks
    - Gray boxes: control logic
    - White ovals: labels for signals
    - Thick lines: 64-bit word values
   - Thin lines: 4-8 bit values
    - Dotted lines: 1-bit values

Fetch Logic
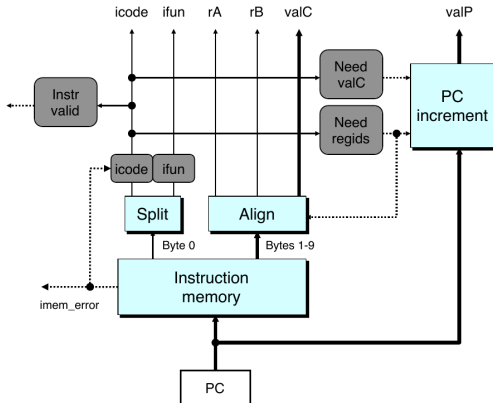
Fetch Logic

- Predefined Blocks
  - PC: Register containing PC
  - Instruction memory: read 10 bytes (PC to PC+9)
    - signal invalid addresses
  - Split: divide instruction byte into `icode` and `ifun`
  - Align: get fields for `rA`, `rB`, and `valC`
- Control Logic
  - Instruction valid: is the instruction valid?
  - `icode`, `ifun`: generate no-op if invalid address
  - Need `regids`: does the instruction have a register byte?
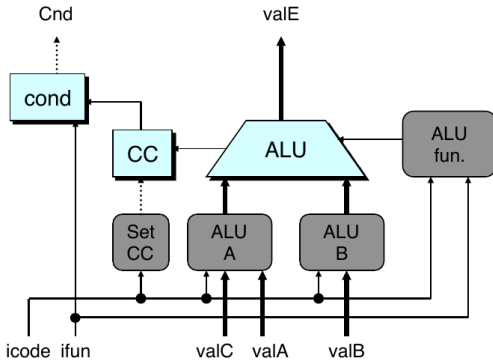 - Need `valC`: does this instruction have a constant word?

Decode Logic
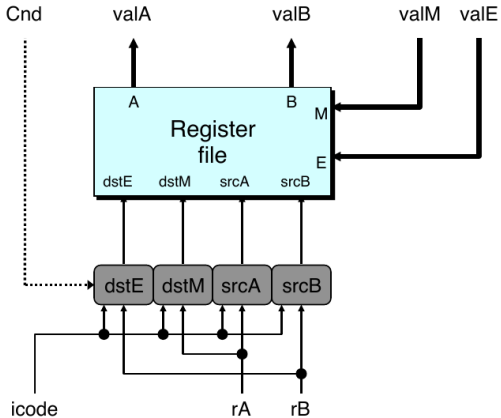
Decode Logic

- Register File
    - Read ports A, B
    - Write ports E, M
    - Addresses are register IDs or 15 (0xF) (no access)
- Control Logic
    - `srcA`, `srcB`: read port addresses
    - `dstE`, `dstM`: write port addresses
- Signals
    - `Cnd`: indicate whether or not to perform conditional move (computed in execute stage)
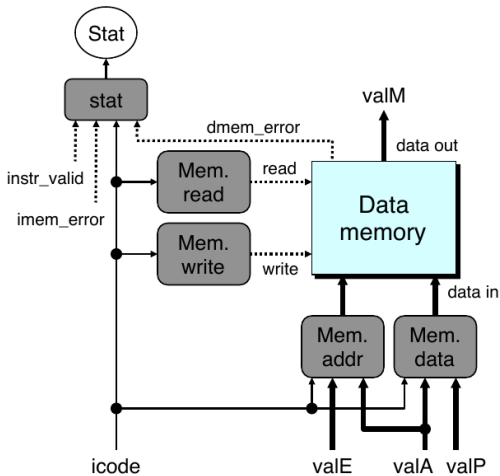
Execute Logic

Execute Logic

- Units
  - ALU: implements 4 required functions and generates condition code values
  - CC: register with 3 condition codes
  - cond: computes conditional jump/move flag
- Control Logic
  - Set CC: should condition code register be loaded?
  - ALU A: input A to ALU
  - ALU B: input B to ALU
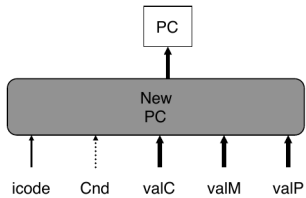 - ALU fun: what function should ALU compute?

Memory Logic

Memory Logic

- Memory
  - Reads or writes memory word
- Control Logic
  - stat: what is instruction status?
  - 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

Sequential Summary

- Implementation
    - Express every instruction as series of simple steps
    - Follow same general flow for each instruction type
    - Assemble registers, memories, pre-designed 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 the clock very slowly
    - Hardware units only active for fraction of clock cycle