

Hardware Control Language

- Very simple hardware description language
- Can only express limited aspects of hardware operation
 - Parts we want to explore and modify

Data Types

- `bool`: Boolean
 - `a, b, c, ...`
- `int`: words
 - `A, B, C, ...`
 - Does not specify word size—bytes, 32-bit words, ...

Statements

- `bool a = bool-expr ;`
- `int A = int-expr ;`

HCL Operations

- Classify by type of value returned

Boolean Expressions

- Logic Operations
 - `a && b, a || b, !a`
- Word Comparisons
 - `A == B, A != B, A < B, A <= B, A >= B, A > B`
- Set Membership
 - `A in { B, C, D }`
 - » Same as `A == B || A == C || A == D`

Word Expressions

- Case expressions
 - `[a : A; b : B; c : C]`
 - Evaluate test expressions `a, b, c, ...` in sequence
 - Return word expression `A, B, C, ...` for first successful test

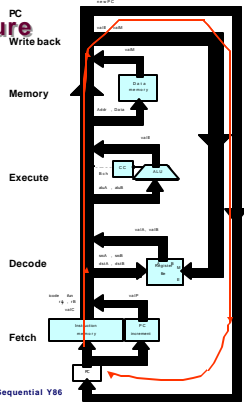
SEQ Hardware Structure

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 data
- Write Back**
 - Write program registers
- PC**
 - Update program counter

CompOrg Fall 2002 Sequential Y86

Instruction Decoding

Instruction Format

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

CompOrg Fall 2002 Sequential Y86

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

CompOrg Fall 2002 Sequential Y86

Executing popl



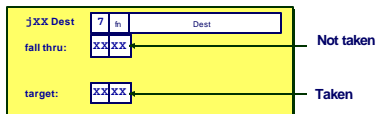
- | | |
|--|--|
| Fetch | Memory |
| <ul style="list-style-type: none"> Read 2 bytes | <ul style="list-style-type: none"> Read from old stack pointer |
| Decode | Write back |
| <ul style="list-style-type: none"> Read stack pointer | <ul style="list-style-type: none"> Update stack pointer Write result to register |
| Execute | PC Update |
| <ul style="list-style-type: none"> Increment stack pointer by 4 | <ul style="list-style-type: none"> Increment PC by 2 |

Stage Computation: popl

	popl rA	
Fetch	code:ifun \rightarrow M[PC] rA:rB \rightarrow M[PC+1] valP \rightarrow PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA \rightarrow R[%esp] valB \rightarrow R[%esp]	Read stack pointer Read stack pointer
Execute	valE \rightarrow valB + 4	Increment stack pointer
Memory	valM \rightarrow M[valA]	Read from stack
Write	R[%esp] \rightarrow valE	Update stack pointer
back	R[rA] \rightarrow valM	Write back result
PC update	PC \rightarrow valP	Update PC

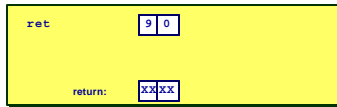
- Use ALU to increment stack pointer
- Must update two registers
 - Popped value
 - New stack pointer

Executing Jumps



- | | |
|--|---|
| Fetch | Memory |
| <ul style="list-style-type: none"> Read 5 bytes Increment PC by 5 | <ul style="list-style-type: none"> Do nothing |
| Decode | Write back |
| <ul style="list-style-type: none"> Do nothing | <ul style="list-style-type: none"> Do nothing |
| Execute | PC Update |
| <ul style="list-style-type: none"> Determine whether to take branch based on jump condition and condition codes | <ul style="list-style-type: none"> Set PC to Dest if branch taken or to incremented PC if not branch |

Executing ret



- | | | | |
|----------------|--|-------------------|--|
| Fetch | <ul style="list-style-type: none"> Read 1 byte | Memory | <ul style="list-style-type: none"> Read return address from old stack pointer |
| Decode | <ul style="list-style-type: none"> Read stack pointer | Write back | <ul style="list-style-type: none"> Update stack pointer |
| Execute | <ul style="list-style-type: none"> Increment stack pointer by 4 | PC Update | <ul style="list-style-type: none"> Set PC to return address |

Stage Computation: ret

	ret	
Fetch	icode:ifun \rightarrow M[PC]	Read instruction byte
Decode	valA \rightarrow R[%esp] valB \rightarrow R[%esp]	Read operand stack pointer Read operand stack pointer
Execute	valE \rightarrow valB + 4	Increment stack pointer
Memory	valM \rightarrow M[valA]	Read return address
Write back	R[%esp] \rightarrow valE	Update stack pointer
PC update	PC \rightarrow valM	Set PC to return address

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

Computation Steps

		OPI rA, rB	
Fetch	icode, ifun	icode:ifun \rightarrow M[PC]	Read instruction byte
	rA, rB	rA::rB \rightarrow M[PC+1]	Read register byte [Read constant word]
	valC	valP \rightarrow PC+2	Compute next PC
	valP		
Decode	valA, srcA	valA \rightarrow R[rA]	Read operand A
	valB, srcB	valB \rightarrow R[rB]	Read operand B
Execute	valE	valE \rightarrow valB OP valA	Perform ALU operation
	Cond code	Set CC	Set condition code register
Memory	valM		[Memory read/write]
Write back	dstE	R[rB] \rightarrow valE	Write back ALU result
back	dstM		[Write back memory result]
PC update	PC	PC \rightarrow valP	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

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

CompOrg Fall 2002 Sequential Y86 25

Fetch Logic

Control Logic

- Instr. Valid: Is this instruction valid?
- Need regids: Does this instruction have a register bytes?
- Need vaIC: Does this instruction have a constant word?

CompOrg Fall 2002 Sequential Y86 26

Fetch Control Logic

```

bool need_regids =
    icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
              IIRMOVL, IRMMOVL, IMRMOVL };

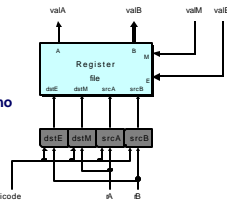
bool instr_valid = icode in
    { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
      IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
  
```

CompOrg Fall 2002 Sequential Y86 27

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
- dstE, dstM: write port addresses

A Source

Decode	OPI rA, rB valA ← R[rA]	Read operand A
Decode	IRMMOVL rA, D(rB) valA ← R[rA]	Read operand A
Decode	POP rA valA ← R[%esp]	Read stack pointer
Decode	JXX Dest	No operand
Decode	CALL Dest	No operand
Decode	RET valA ← R[%esp]	Read stack pointer

```
int srcA = [
  icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
  icode in { IPOPL, IRET } : RESP;
  1 : RNONE; # Don't need register
];
```

E Destination

Write-back	OPI rA, rB R[rB] ← valE	Write back result
Write-back	IRMMOVL rA, D(rB)	None
Write-back	POP rA R[%esp] ← valE	Update stack pointer
Write-back	JXX Dest	None
Write-back	CALL Dest R[%esp] ← valE	Update stack pointer
Write-back	RET R[%esp] ← valE	Update stack pointer

```
int dstE = [
  icode in { IRRMOVL, IRMMOVL, IOPL } : rB;
  icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
  1 : RNONE; # Don't need register
];
```

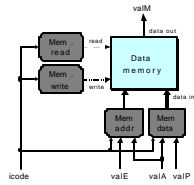
Memory Logic

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

Memory	OP rA, rB	No operation
Memory	rmmovl rA, D(rB)	Write value to memory
Memory	popl rA	Read from stack
Memory	valM \leftarrow M[valA]	
Memory	jXX Dest	No operation
Memory	call Dest	
Memory	M[valE] \leftarrow valP	Write return value on stack
Memory	ret	
Memory	valM \leftarrow M[valA]	Read return address

```
int mem_addr = {
  icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
  icode in { IPOPL, IRET } : valA;
  # Other instructions don't need address
};
```

Memory Read

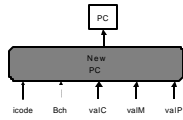
Memory	OP rA, rB	No operation
Memory	rmmovl rA, D(rB)	Write value to memory
Memory	popl rA	Read from stack
Memory	valM \leftarrow M[valA]	
Memory	jXX Dest	No operation
Memory	call Dest	
Memory	M[valE] \leftarrow valP	Write return value on stack
Memory	ret	
Memory	valM \leftarrow M[valA]	Read return address

```
bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
```

PC Update Logic

New PC

- Select next value of PC

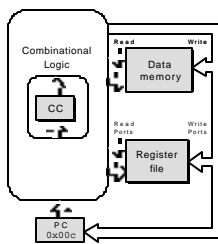


PC Update

PC update	OPI rA, rB	
PC update	PC ← valP	Update PC
	rmmovl rA, D(rB)	
PC update	PC ← valP	Update PC
	popl rA	
PC update	PC ← valP	Update PC
	JXX Dest	
PC update	PC ← Bch ? valC : valP	Update PC
	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

- PC register
 - Cond. Code register
 - Data memory
 - Register file
- All updated as clock rises

Combinational Logic

- ALU
- Control logic
- Memory reads
 - Instruction memory
 - Register file
 - Data memory

SEQ Operation #2

Cycle	Instruction	PC	CC
1	0x0001 irmovl \$0x100,%ebx # 1 ebx <-- 0x100	0x0000	000
2	0x0061 irmovl \$0x200,%ebx # 1 ebx <-- 0x200	0x0001	000
3	0x00c1 addl %ebx,%ebx # 1 ebx <-- 0x300 CC <-- 000	0x0061	000
4	0x00e1 je %ebx # NOT taken	0x00c1	000

- state set according to second `irmovl` instruction
- combinational logic starting to react to state changes

CompOrg Fall 2002 Sequential Y86 40

SEQ Operation #3

Cycle	Instruction	PC	CC
1	0x0001 irmovl \$0x100,%ebx # 1 ebx <-- 0x100	0x0000	000
2	0x0061 irmovl \$0x200,%ebx # 1 ebx <-- 0x200	0x0001	000
3	0x00c1 addl %ebx,%ebx # 1 ebx <-- 0x300 CC <-- 000	0x0061	000
4	0x00e1 je %ebx # NOT taken	0x00c1	000

- state set according to second `irmovl` instruction
- combinational logic generates results for `addl` instruction

CompOrg Fall 2002 Sequential Y86 41

SEQ Operation #4

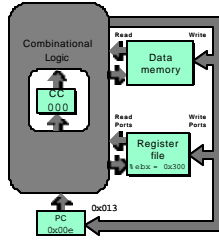
Cycle	Instruction	PC	CC
1	0x0001 irmovl \$0x100,%ebx # 1 ebx <-- 0x100	0x0000	000
2	0x0061 irmovl \$0x200,%ebx # 1 ebx <-- 0x200	0x0001	000
3	0x00c1 addl %ebx,%ebx # 1 ebx <-- 0x300 CC <-- 000	0x0061	000
4	0x00e1 je %ebx # NOT taken	0x00c1	000

- state set according to `addl` instruction
- combinational logic starting to react to state changes

CompOrg Fall 2002 Sequential Y86 42

SEQ Operation #5

Clock	Cycle 1	Cycle 2	Cycle 3	Cycle 4
Op	0x0001	0x0002	0x0100, 0x02	0x1000
Op	0x0006	0x0002	0x0200, 0x02	0x2000
Op	0x000c	addl %eax, %eax	0x1000	CC ← 000
Op	0x000e	je %eax	0x0000	Branch taken



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

CompOrg Fall 2002 Sequential Y86

43

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

CompOrg Fall 2002 Sequential Y86

44
