

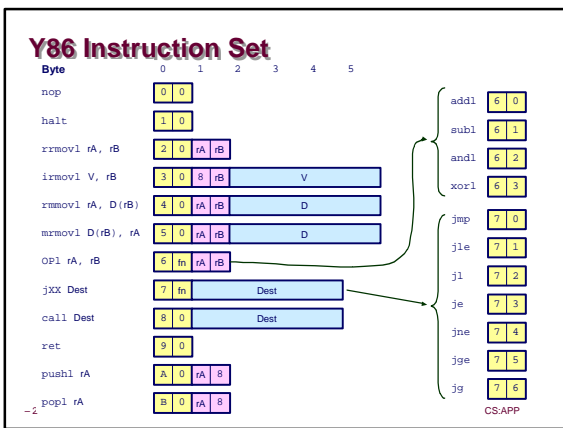
CS:APP Chapter 4
Computer Architecture
Sequential
Implementation

Randal E. Bryant

Carnegie Mellon University

<http://csapp.cs.cmu.edu>

CS:APP



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

CS:APP

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 ;`

-4- CS:APP

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

-5- CS:APP

SEQ Hardware Structure

The diagram illustrates the SEQ hardware structure. It shows a vertical pipeline with four main stages: Fetch, Decode, Execute, and Memory.

- Fetch:** The Program Counter (PC) provides the address for the Instruction Memory. The Instruction Memory outputs the instruction (Inst). The PC is updated by the PC Controller.
- Decode:** The instruction is decoded by the Register File (RF) to determine the Register Address (RA) and Register File Address (RFA). The ALU performs operations on RA and RFA.
- Execute:** The ALU result is used to calculate the Memory Address (MA) and Memory File Address (MFA). The Memory File outputs the Memory Data (MD).
- Memory:** The Memory Data is used to calculate the Write Back (WB) value. The WB value is then used to update the PC.

 A red arrow indicates the flow of the instruction through the stages, and a black arrow shows the flow of data and control signals.

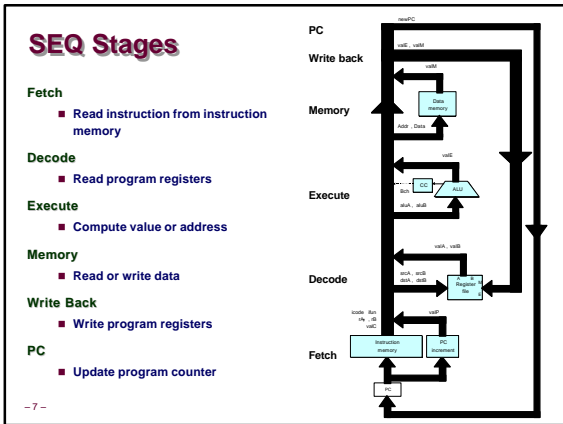
State

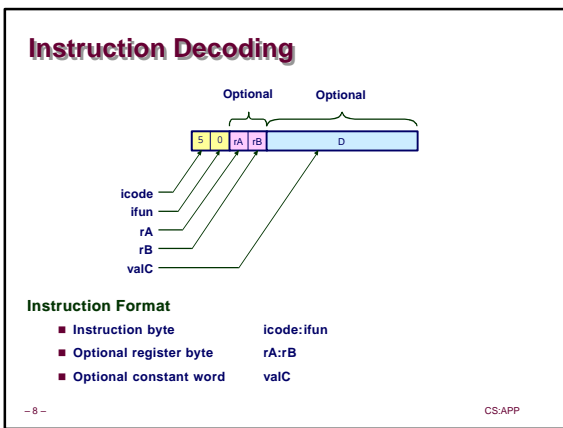
- Program counter register (PC)
- Condition code register (CC)
- Register File
- Memories
 - Access same memory space
 - Data: for reading/writing program data
 - Instruction: for reading instructions

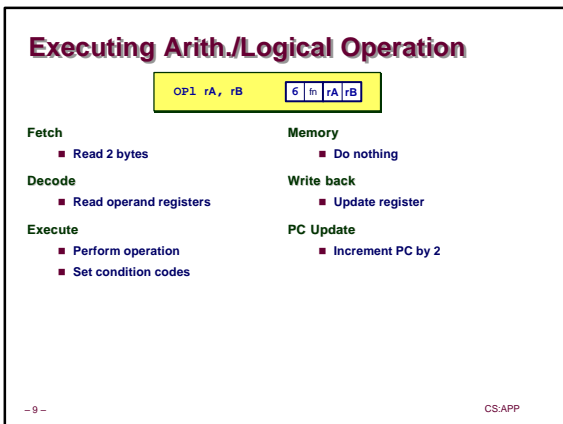
Instruction Flow

- Read instruction at address specified by PC
- Process through stages
- Update program counter

-6- CS:APP







Stage Computation: Arith/Log. Ops

	OPI rA, rB	
Fetch	icode:ifun \leftarrow M ₁ [PC] rA:rB \leftarrow M ₂ [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 sequence of simple steps
- Use same general form for all instructions

- 10 - CS:APP

Executing rmmovl

rmmovl rA, D(rB) 4 0 rA rB D

Fetch	Memory
<ul style="list-style-type: none"> Read 6 bytes 	<ul style="list-style-type: none"> Write to memory
Decode	Write back
<ul style="list-style-type: none"> Read operand registers 	<ul style="list-style-type: none"> Do nothing
Execute	PC Update
<ul style="list-style-type: none"> Compute effective address 	<ul style="list-style-type: none"> Increment PC by 6

- 11 - CS:APP

Stage Computation: rmmovl

	rmmovl rA, D(rB)	
Fetch	icode:ifun \leftarrow M ₁ [PC] rA:rB \leftarrow M ₂ [PC+1] valC \leftarrow M ₃ [PC+2] valP \leftarrow PC+6	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	M ₄ [valE] \leftarrow valA	Write value to memory
Write back		
PC update	PC \leftarrow valP	Update PC

- Use ALU for address computation

- 12 - CS:APP

Executing popl

popl rA b 0 rA 8

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

- 13 - CS:APP

Stage Computation: popl

	popl rA icode:ifun ← M ₁ [PC] rA:rB ← M ₂ [PC+1]	Read instruction byte Read register byte
Fetch	valP ← PC+2	Compute next PC Read stack pointer
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	R[%esp] ← valE	Update stack pointer
back	R[rA] ← valM	Write back result
PC update	PC ← valP	Update PC

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

- 14 - CS:APP

Executing Jumps

jXX Dest	7	in	Dest	
fall thru:	XX XX			Not taken
target:	XX XX			Taken

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

- 15 - CS:APP

Stage Computation: Jumps

	<p>Jump Dest</p> <p>icode:ifun $\leftarrow M_1[PC]$</p> <p>valC $\leftarrow M_4[PC+1]$</p> <p>valP $\leftarrow PC+5$</p>	<p>Read instruction byte</p> <p>Read destination address</p> <p>Fall through address</p>
Fetch		
Decode		
Execute	Bch $\leftarrow \text{Cond}(CC, ifun)$	Take branch?
Memory		
Write back		
PC update	PC $\leftarrow \text{Bch} ? \text{valC} : \text{valP}$	Update PC

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

- 16 - CS:APP

Executing call

call Dest: 8 0 Dest

return: XX XX

target: XX XX

<p>Fetch</p> <ul style="list-style-type: none"> ■ Read 5 bytes ■ Increment PC by 5 <p>Decode</p> <ul style="list-style-type: none"> ■ Read stack pointer <p>Execute</p> <ul style="list-style-type: none"> ■ Decrement stack pointer by 4 	<p>Memory</p> <ul style="list-style-type: none"> ■ Write incremented PC to new value of stack pointer <p>Write back</p> <ul style="list-style-type: none"> ■ Update stack pointer <p>PC Update</p> <ul style="list-style-type: none"> ■ Set PC to Dest
--	--

- 17 - CS:APP

Stage Computation: call

	<p>call Dest</p> <p>icode:ifun $\leftarrow M_1[PC]$</p> <p>valC $\leftarrow M_4[PC+1]$</p> <p>valP $\leftarrow PC+5$</p>	<p>Read instruction byte</p> <p>Read destination address</p> <p>Compute return point</p>
Fetch		
Decode	valB $\leftarrow R[\%esp]$	Read stack pointer
Execute	valE $\leftarrow \text{valB} + -4$	Decrement stack pointer
Memory	$M_4[\text{valE}] \leftarrow \text{valP}$	Write return value on stack
Write back	$R[\%esp] \leftarrow \text{valE}$	Update stack pointer
PC update	PC $\leftarrow \text{valC}$	Set PC to destination

- Use ALU to decrement stack pointer
- Store incremented PC

- 18 - CS:APP

Executing ret

ret 9 0

return: xx|xx

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

- 19 - CS:APP

Stage Computation: ret

Fetch	icode:ifun ← M ₁ [PC]	Read instruction byte
Decode	valA ← R[%esp] valB ← R[%esp]	Read operand stack pointer Read operand stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M ₁ [valA]	Read return address
Write	R[%esp] ← valE	Update stack pointer
back		
PC update	PC ← valM	Set PC to return address

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

- 20 - CS:APP

Computation Steps

	OPI rA, rB		
Fetch	icode,ifun	icode:ifun ← M ₁ [PC]	Read instruction byte
	rA,rB	rA:rB ← M ₁ [PC+1]	Read register byte [Read constant word]
	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	dstE	R[rB] ← valE	Write back ALU result
back	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

- 21 - CS:APP

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

- 25 - CS:APP

Fetch Logic

Control Logic

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

- 26 - CS:APP

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 };
    
```

- 27 - CS:APP

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

-28 - CS:APP

A Source

Decode	OPl rA, rB valA ← R[rA]	Read operand A
Decode	xmmovl rA, D(rB) valA ← R[rA]	Read operand A
Decode	popl 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, IIRMOVL, IOPL, IPUSHL } : rA;
  icode in { IPOPL, IRET } : RESP;
  1 : RNONE; # Don't need register
};
```

-29 - CS:APP

E Destination

Write-back	OPl rA, rB R[rB] ← valE	Write back result
Write-back	xmmovl rA, D(rB)	None
Write-back	popl 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, IIRMOVL, IOPL } : rB;
  icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
  1 : RNONE; # Don't need register
};
```

-30 - CS:APP

Execute Logic

Units

- ALU**
 - Implements 4 required 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?

- 31 - CS:APP

ALU A Input

Execute	<code>OPl rA, rB</code>	Perform ALU operation
Execute	<code>valE ← valB OP valA</code>	
Execute	<code>rmmovl rA, D(rB)</code>	Compute effective address
Execute	<code>valE ← valB + valC</code>	
Execute	<code>popl rA</code>	Increment stack pointer
Execute	<code>valE ← valB + 4</code>	
Execute	<code>JX Dest</code>	No operation
Execute	<code>call Dest</code>	Decrement stack pointer
Execute	<code>valE ← valB + -4</code>	
Execute	<code>ret</code>	Increment stack pointer
Execute	<code>valE ← valB + 4</code>	

```

int aluA = [
  icode in { IRRMOVL, IOPL } : valA;
  icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
  icode in { ICALL, IPUSHL } : -4;
  icode in { IRET, IPOPL } : 4;
  # Other instructions don't need ALU
];
    
```

- 32 - CS:APP

ALU Operation

Execute	<code>OPl rA, rB</code>	Perform ALU operation
Execute	<code>valE ← valB OP valA</code>	
Execute	<code>rmmovl rA, D(rB)</code>	Compute effective address
Execute	<code>valE ← valB + valC</code>	
Execute	<code>popl rA</code>	Increment stack pointer
Execute	<code>valE ← valB + 4</code>	
Execute	<code>JX Dest</code>	No operation
Execute	<code>call Dest</code>	Decrement stack pointer
Execute	<code>valE ← valB + -4</code>	
Execute	<code>ret</code>	Increment stack pointer
Execute	<code>valE ← valB + 4</code>	

```

int alufun = [
  icode == IOPL : ifun;
  1 : ALUADD;
];
    
```

- 33 - CS:APP

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

The diagram shows a central 'Data memory' block. It has two control inputs: 'Mem.read' and 'Mem.write'. It has two address inputs: 'Mem.addr.' and 'Mem.data.'. It has two data outputs: 'valM' and 'data out'. The 'Mem.read' and 'Mem.write' inputs are connected to a control logic block. The 'Mem.addr.' and 'Mem.data.' inputs are connected to a control logic block. The 'valM' and 'data out' outputs are connected to a control logic block.

- 34 - CS:APP

Memory Address

Memory	OPl rA, rB	No operation
Memory	rmmovl rA, D(rB)	Write value to memory
Memory	popl rA	Read from stack
Memory	jXX Dest	No operation
Memory	call Dest	Write return value on stack
Memory	ret	Read return address

```

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

- 35 - CS:APP

Memory Read

Memory	OPl rA, rB	No operation
Memory	rmmovl rA, D(rB)	Write value to memory
Memory	popl rA	Read from stack
Memory	jXX Dest	No operation
Memory	call Dest	Write return value on stack
Memory	ret	Read return address

```

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

- 36 - CS:APP

PC Update Logic

New PC

- Select next value of PC

PC
 New PC
 icode Bch valC valM valP

- 37 - CS:APP

PC Update

PC update	OPl rA, rB PC ← valP	Update PC
PC update	rmmovl rA, D(rB) PC ← valP	Update PC
PC update	popl rA PC ← valP	Update PC
PC update	jXX Dest PC ← Bch ? valC : valP	Update PC
PC update	call Dest PC ← valC	Set PC to destination
PC update	ret PC ← valM	Set PC to return address

```

int new_pc = [
    icode == ICALL : valC;
    icode == IJXX && Bch : valC;
    icode == IRET : valM;
    1 : valP;
];
    
```

- 38 - CS:APP

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

- 39 - CS:APP

SEQ Operation #2

Clock	Instruction	PC	Register file	CC
Cycle 1	0x0001 irmovl \$0x100,%ebx # %ebx <-- 0x100	0x0001	%ebx = 0x100	000
Cycle 2	0x0006 irmovl \$0x200,%edx # %edx <-- 0x200	0x0006	%edx = 0x200	000
Cycle 3	0x000c addl %edx,%ebx # %ebx <-- 0x300 CC <-- 000	0x000c	%ebx = 0x100	100
Cycle 4	0x0004 je %ebx # NOT taken	0x000c	%ebx = 0x100	100

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

-40 - CS:APP

SEQ Operation #3

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

-41 - CS:APP

SEQ Operation #4

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

-42 - CS:APP

SEQ Operation #5

Cycle	Instruction	imm	Op	Op2	Op3	Op4	Op5	Op6	Op7	Op8	Op9	Op10
1	0x0001	imm=0	0x100	#ebx	#	ebx	<-<	0x100				
2	0x0005	imm=0	0x200	#ebx	#	ebx	<-<	0x200				
3	0x000c	addl	%ebx	%ebx	#	ebx	<-<	0x300	CC	<-<	000	
4	0x000e	je	%eax	#	000	%eax						

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

- 43 - CS:APP

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

- 44 - CS:APP
