Lecture 2: Instruction Set Architecture

**ECE 411 COMPUTER ORGANIZATION AND DESIGN**

Instruction Set Architecture (ISA)

● serves as an interface b/w software and hardware

● provides a mechanism by which the software tells the hardware what should be done

high level language code : C, C++, Java, Fortran,
compiler
assembly language code: architecture specific statements
assembler
machine language code: architecture specific bit patterns

SW

ISA

HW

ECE 411 COMPUTER ORGANIZATION AND DESIGN
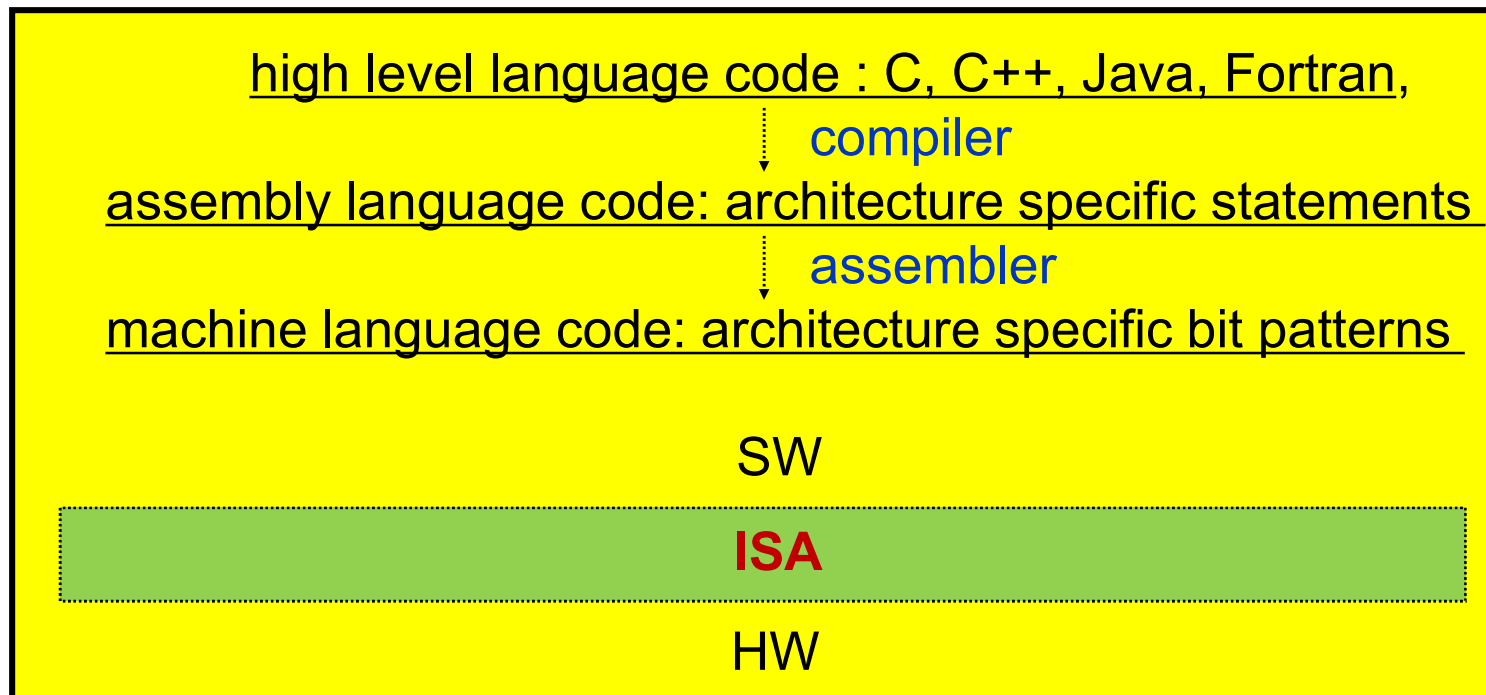
Interface Design

- a good interface:
  - ✓ lasts through many implementations (portability, compatability)
  - ✓ is used in many different ways (generality)
  - ✓ provides convenient functionality to higher levels
  - ✓ permits an efficient implementation at lower levels

```
use ───┐
        ├──→ interface ──→ imp 1
use ───┤                   imp 2
        │
use ───┘                   imp 3
```

time ↓

ECE 411 COMPUTER ORGANIZATION AND DESIGN
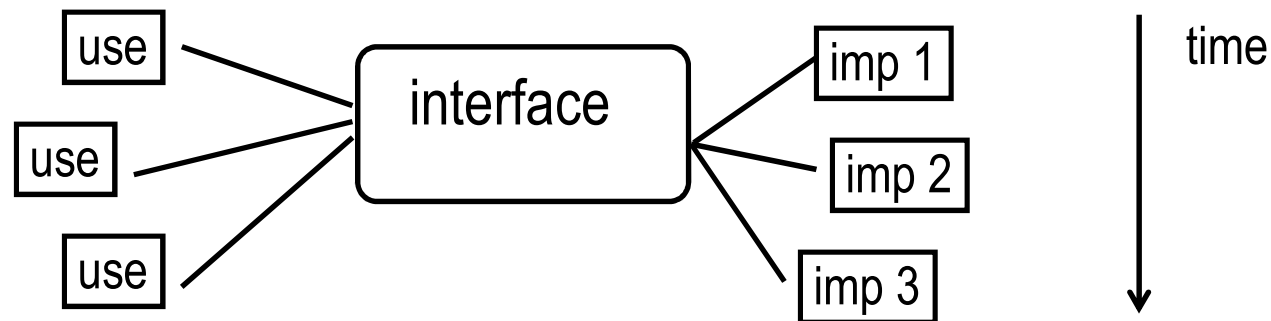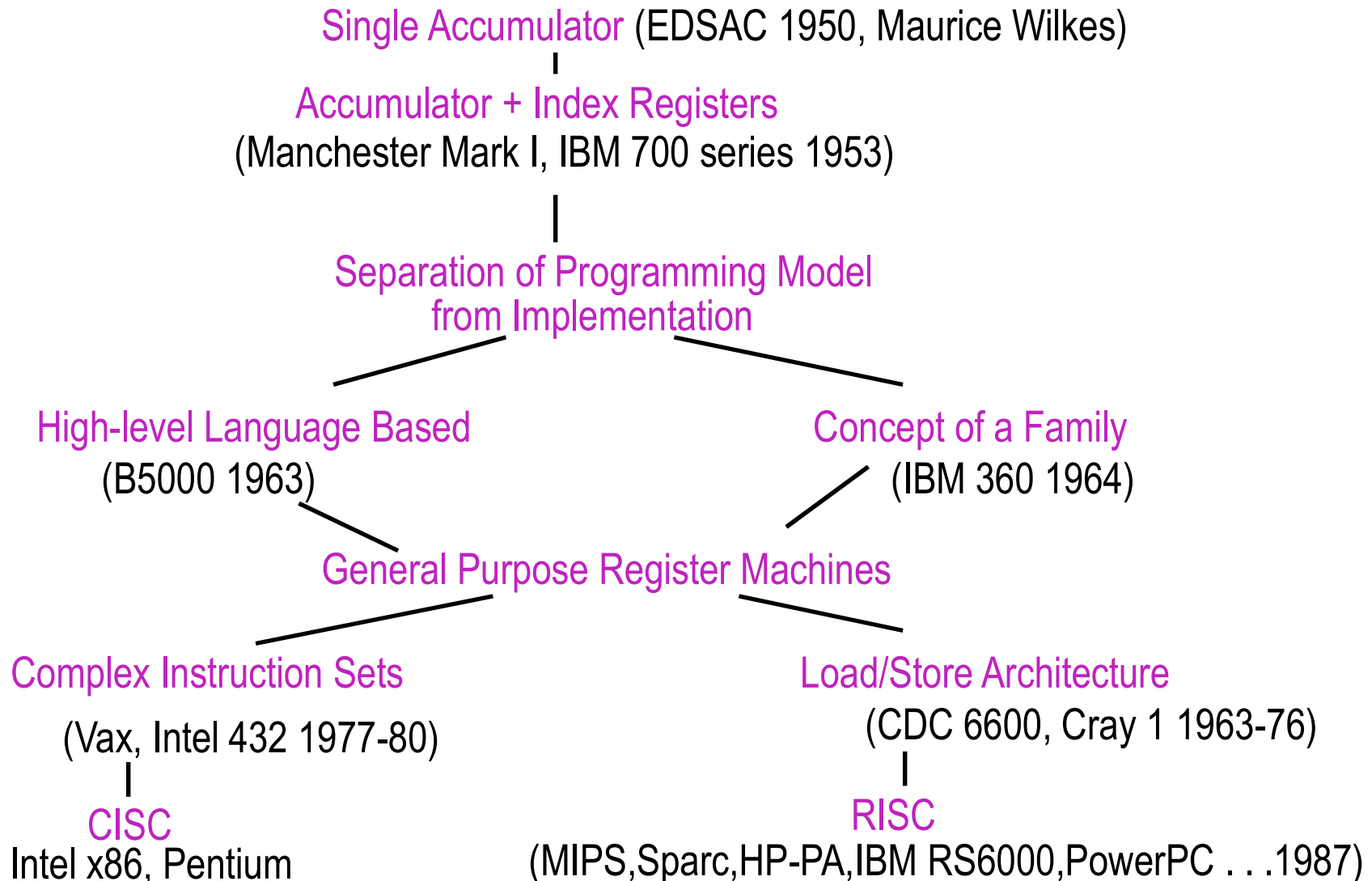
Instruction Set Design Issues

- instruction set design issues include:
  - ✓ where are operands stored?
    - ○ registers, memory, stack, accumulator
  - ✓ how many explicit operands are there?
    - ○ 0, 1, 2, or 3
  - ✓ how is the operand location specified?
    - ○ register, immediate, indirect, . . .
  - ✓ what type & size of operands are supported?
    - ○ byte, int, float, double, string, vector. . .
  - ✓ what operations are supported?
    - ○ add, sub, mul, move, compare . . .

$ADD \quad R1 \quad R2$

$R1 \leftarrow R_1 + R_2$

$ADD \quad R_3 \quad R_1 \quad R_2$

$R_3 \leftarrow R_1 + R_2$

$ADD \quad R_3 \quad R_1 \quad [R_2]$

$R_3 \leftarrow R_1 + MEM[R_2]$

Evolution of Instruction Sets

Single Accumulator (EDSAC 1950, Maurice Wilkes)

Accumulator + Index Registers
(Manchester Mark I, IBM 700 series 1953)

Separation of Programming Model
from Implementation

High-level Language Based
(B5000 1963)

Concept of a Family
(IBM 360 1964)

General Purpose Register Machines

Complex Instruction Sets

(Vax, Intel 432 1977-80)

CISC
Intel x86, Pentium

Load/Store Architecture

(CDC 6600, Cray 1 1963-76)

RISC
(MIPS,Sparc,HP-PA,IBM RS6000,PowerPC . . .1987)

Instruction Length

variable:

x86 – Instructions vary from 1 to 17 Bytes long

VAX – from 1 to 54 Bytes

fixed:

MIPS, PowerPC, and most other RISC's:

all instruction are 4 Bytes long

ECE 411 COMPUTER ORGANIZATION AND DESIGN

Instruction Length

- variable-length instructions (x86, VAX):
  - ✓ require multi-step, complex fetch and decode (-)
  - ✓ allow smaller binary programs that require less disk storage, less DRAM at runtime, less memory, bandwidth and better cache efficiency (+)

- fixed-length instructions (RISC's)
  - ✓ allow easy fetch and decode (+)
  - ✓ simplify pipelining and parallelism (+)
  - ✓ result in larger binary programs that require more disk storage, more DAM at runtime, more memory bandwidth and lower cache efficiency (-)

DRAM

Why do we care about memory or disk space.

Magnetic Core Memory (1955-1975)
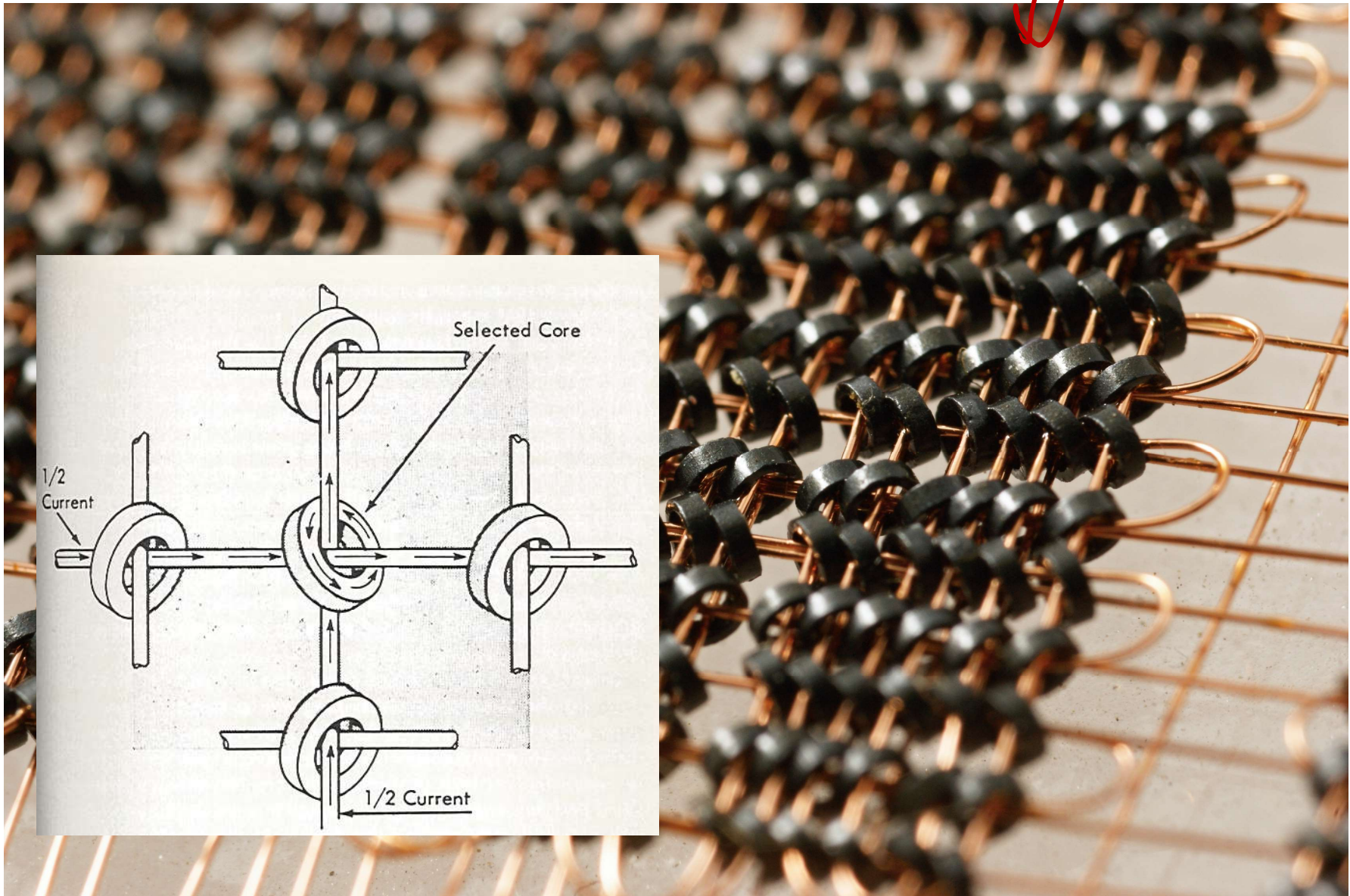
ARM Case Study

- ARM (Advanced RISC Machine)
  - ✓ started with fixed, 32-bit instruction length
  - ✓ added thumb instructions
    - o a subset of the 32-bit instructions
    - o all encoded in 16 bits
    - o all translated into equivalent 32-bit instructions within the processor pipeline at runtime
    - o can access only 8 general purpose registers
  - ✓ motivated by many resource constrained embedded applications that require less disk storage, less DRAM at runtime, less memory bandwidth and better cache efficiency

How many registers?

- most computers have a small set of registers
  - ✓ on-chip memory to hold values that will be used soon
  - ✓ a typical instruction use 2 or 3 register values

- advantages of a small number of registers:
  - ✓ it requires fewer instruction bits to specify which one
  - ✓ less hardware
  - ✓ faster access (shorter wires, fewer gates)
  - ✓ faster context switch (when all registers need saving)

- advantages of a larger number:
  - ✓ fewer loads and stores needed
  - ✓ easier to express several operations in parallel

$R1 \leftarrow R_2 + R_3$

$R_4 \leftarrow R_1 + R_3$

$R_5 \leftarrow R_4 + R_6$

In 411, "load" means moving data from memory to register, "store" is reverse

Where do operands reside?

when the ALU needs them?

- stack machine
  - ✓ push loads memory into 1st register ("top of stack"), moves other regs down
  - ✓ pop does the reverse
  - ✓ add combines contents of first two regs, moves rest up

- accumulator machine:
  - ✓ only 1 register (called the "accumulator")
  - ✓ instruction include store and ACC ← ACC + MEM

- register-memory machine :
  - ✓ arithmetic instrs can use data in registers and/or memory

- load-store machine (aka register-register machine):
  - ✓ arithmetic instructions can only use data in registers

Classifying ISAs

accumulator (before 1960, e.g. 68HC11):

1-address   add A   acc ← acc + mem[A]

stack (1960s to 1970s):

0-address   add   tos ← tos + next

memory-Memory (1970s to 1980s):

2-address   add A B   mem[A] ← mem[A] + mem[B]

3-address   add A, B, C   mem[A] ← mem[B] + mem[C]

register-memory (1970s to present, e.g. 80x86):

2-address   add R1, A   R1 ← R1 + mem[A]

                   load R1, A   R1 ← mem[A]

register-register (load/store) (1960s to present, e.g. MIPS):

3-address   add R1, R2, R3   R1 ← R2 + R3

                   load R1, R2   R1 ← mem[R2]

                   store R1, R2   mem[R1] ← R2

ECE 411 COMPUTER ORGANIZATION AND DESIGN

Operand Locations in Four ISA Classes

(a) Stack
(b) Accumulator
(c) Register-memory
(d) Register-register/load-store

GPR

Processor

TOS
ALU
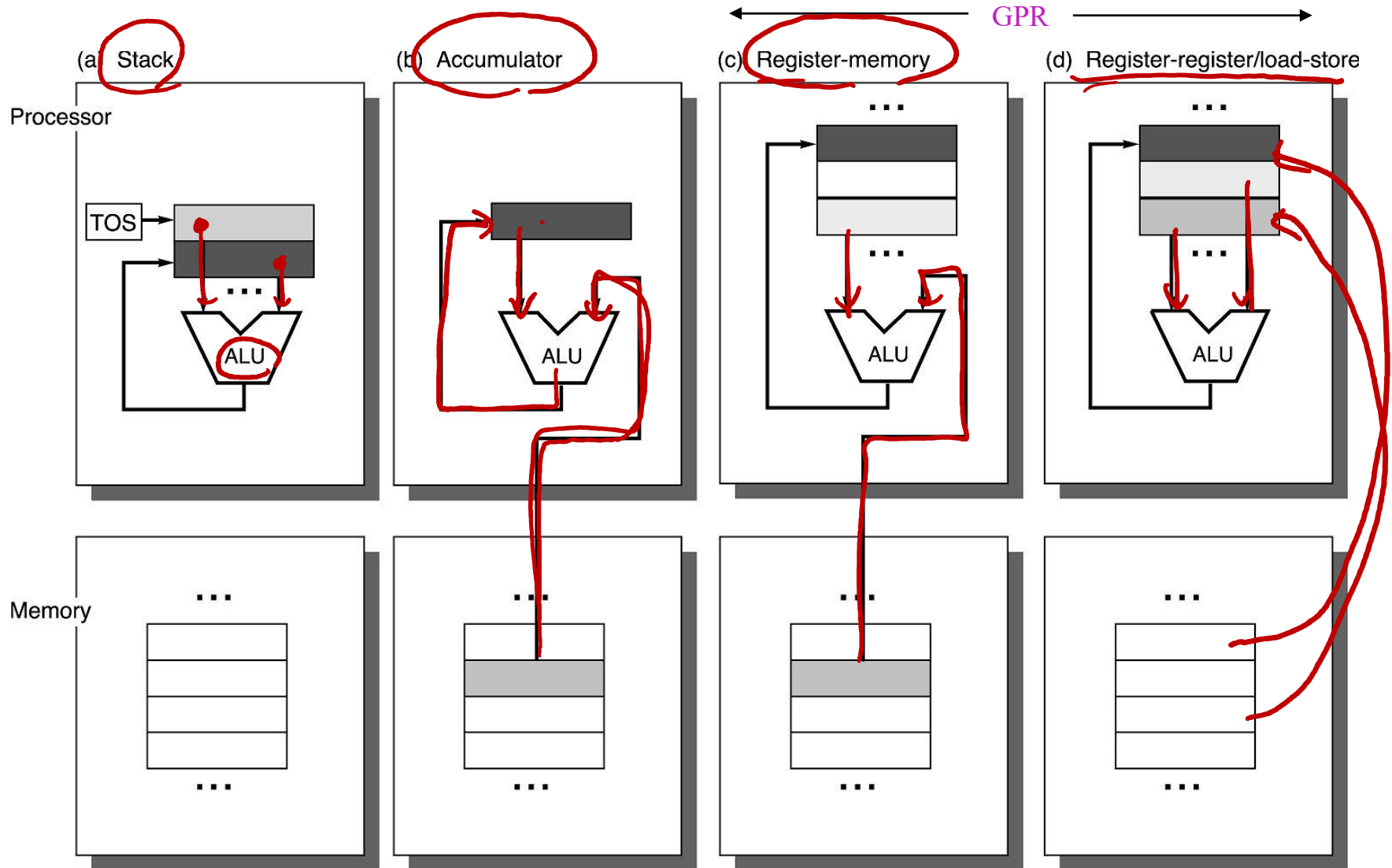
Memory

Comparing the ISA classes

code sequence for **C = A + B**

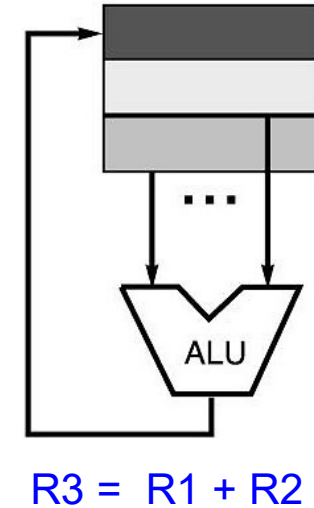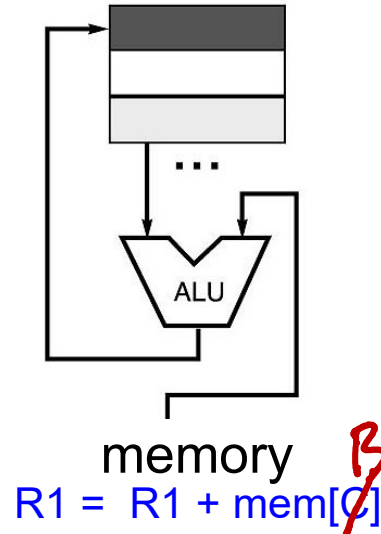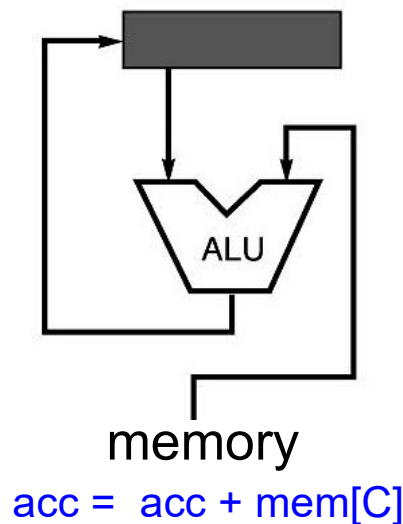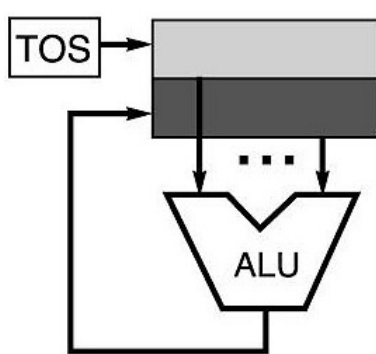| stack	accumulator	register-memory	load-store
`Push A`	`Load   A`	`Add C, A, B`	`Load   R1,A`
`Push B`	`Add    B`		`Load   R2,B`
`Add`	`Store C`		`Add    R3,R1,R2`
`Pop  C`			`Store C,R3`
Java VMs	DSPs	VAX, x86 partially	

Four Instruction Sets

- code sequence C = A + B

stack	accumulator	register (register-memory)	register (load-store)
push A	load A	load R1,A	load R1,A
push B	add B	add R1,B	load R2,B
add	store C	store C, R1	add R3, R1, R2
pop C			store C, R3

Handwritten annotations: ACC ← MEM[A], ACC + MEM[B], R1 ← R1 + MEM[B], MEM[A]

acc = acc + mem[C]   R1 = R1 + mem[C]   R3 = R1 + R2

ECE 411 COMPUTER ORGANIZATION AND DESIGN

Four Instruction Sets

- A = X*Y + X*Z

Stack	Accumulator	Register (register-memory)	Register (load-store)

Stack

Accumulator

R1 ☐
R2 ☐
R3 ☐

Memory

|	
A	?
X	12
Y	3
B	4
C	5
temp	?

More About General Purpose Registers

- why do almost all new architectures use GPRs?
  - ✓ registers are much faster than memory (even cache)
    - register values are available immediately
    - when memory isn't ready, processor must wait ("stall")
  - ✓ registers are convenient for variable storage
    - compiler assigns some variables just to registers
    - more compact code since small fields specify registers (compared to memory addresses)
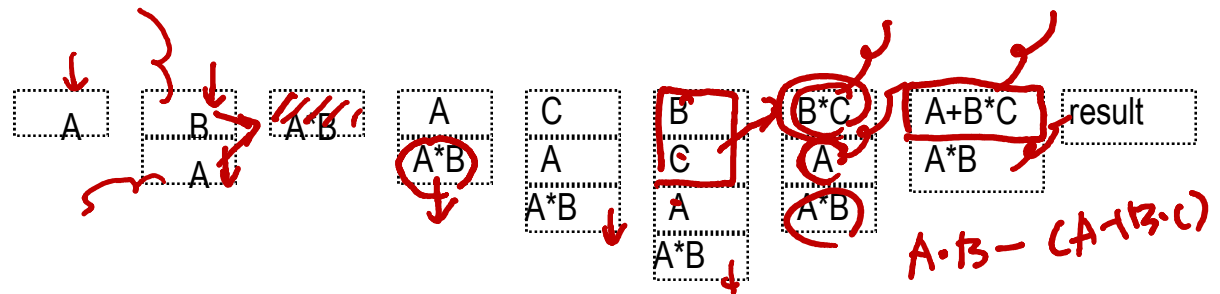
SWAP (A, B);

tmp ← A
A ← B
B ← tmp

processor · memory · disk

registers · cache
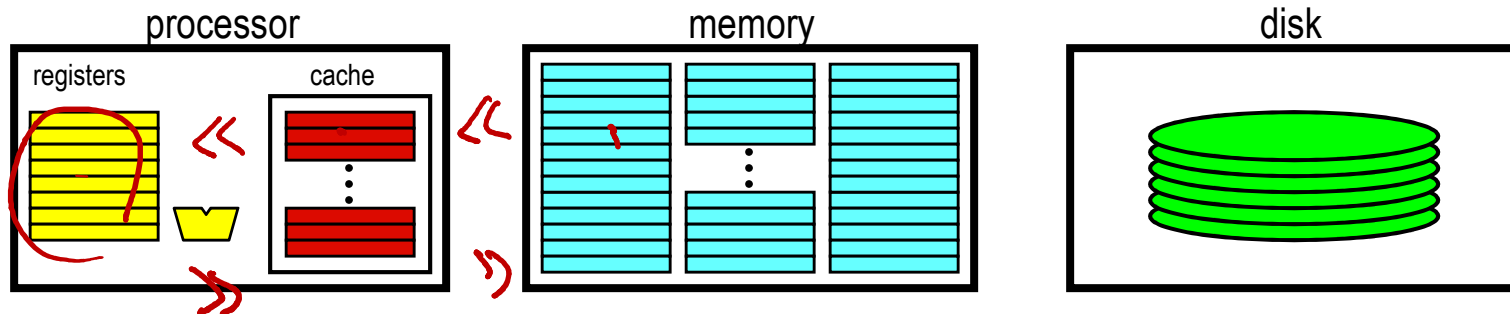
Stack Architectures

- instruction set:
  - ✓ add, sub, mult, div, . . .
  - ✓ push A, pop A

- example: A*B - (A+C*B)
  - ✓ push A
  - ✓ push B
  - ✓ mul
  - ✓ push A
  - ✓ push C
  - ✓ push B
  - ✓ mul
  - ✓ add
  - ✓ sub

TOS → [ALU]

A	B	A·B	A	C	B	B*C	A+B*C	result
	A	A*B	A*B	A	C	A	A*B	
				A*B	A	A	A*B	
					A*B	A*B		

A·B — (A+B·C)
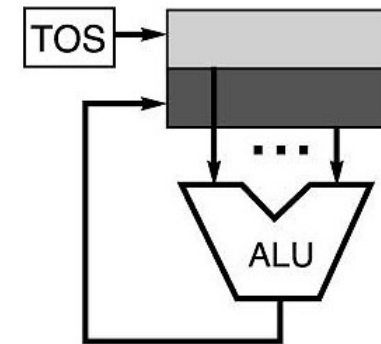
Stacks: Pros and Cons

- pros
  - ✓ good code density (implicit top of stack)
  - ✓ low hardware requirements
  - ✓ easy to write a simpler compiler for stack architectures

- cons
  - ✓ stack becomes the bottleneck
  - ✓ little ability for parallelism or pipelining
  - ✓ data is not always at the top of stack when need, so additional instructions like top and swap are needed
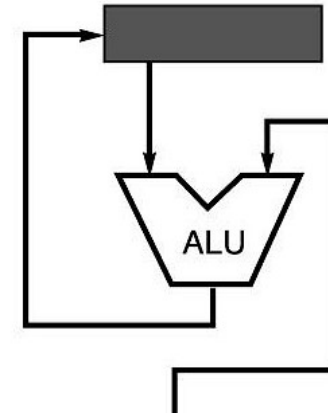 - ✓ difficult to write an optimizing compiler for stack architectures

Accumulator Architectures

- instruction set:
  - ✓ add A, sub A, mult A, div A, . . .
  - ✓ load A, store A

- example: A*B - (A+C*B)
  - ✓ load B
  - ✓ mul C
  - ✓ add A
  - ✓ store D
  - ✓ load A
  - ✓ mul B
  - ✓ sub D

acc = acc +,-,*,/ mem[A]

B	B*C	A+B*C	A+B*C	A	A*B	result

ECE 411 COMPUTER ORGANIZATION AND DESIGN
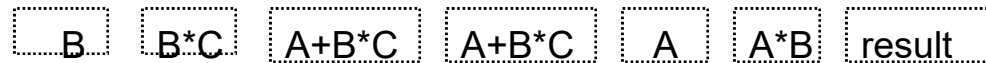
Accumulators: Pros and Cons

- pros
  - ✓ very low hardware requirements
  - ✓ easy to design and understand

- cons
  - ✓ accumulator becomes the bottleneck
  - ✓ little ability for parallelism or pipelining
  - ✓ high memory traffic

Memory-Memory Architectures

- instruction set:

  (3 operands)   add A, B, C   sub A, B, C   mul A, B, C

  (2 operands)   add A, B   sub A, B mul A, B

- example: A*B - (A+C*B)

  (3 operands)

  mul D, A, B
  mul E, C, B
  add E, A, E
  sub E, D, E

  (2 operands)
  mov D, A
  mul D, B
  mov E, C
  mul E, B
  add E, A
  sub E, D

Memory-Memory: Pros and Cons

- pros
  - ✓ requires fewer instructions (especially if 3 operands)
  - ✓ easy to write compilers for (especially if 3 operands)

- cons
  - ✓ very high memory traffic (especially if 3 operands)
  - ✓ variable number of clocks per instruction
  - ✓ w/ two operands, more data movements are required

Register-Memory Architectures

- instruction set:
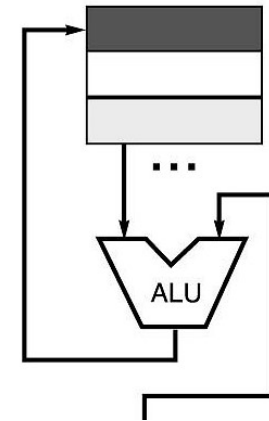
  add R1, A   sub R1, A   mul R1, B

  load R1, A   store R1, A

- example: A*B - (A+C*B)

  load R1, A

  mul R1, B   /*   A*B   */

  store R1, D

  load R2, C

  mul R2, B   /*   C*B   */

  add R2, A   /*   A + CB   */

  sub R2, D   /*   AB - (A + C*B) */

R1 = R1 +,-,*,/ mem[B]

Memory-Register: Pros and Cons

- pros
  - ✓ some data can be accessed w/o loading first
  - ✓ instruction format easy to encode
  - ✓ good code density

- cons
  - ✓ operands are not equivalent (poor orthogonal)
  - ✓ variable number of clocks per instruction
  - ✓ may limit number of registers

Load-Store Architectures

- instruction set:

  add R1, R2, R3   sub R1, R2, R3   mul R1, R2, R3

  load R1, A   store R1, A   move R1, R2

- example: A*B - (A+C*B)

  load R1, A

  load R2, B

  load R3, C

  mul R7, R3, R2   /*   C*B   */

  add R8, R7, R1   /*   A + C*B   */

  mul R9, R1, R2   /*   A*B   */
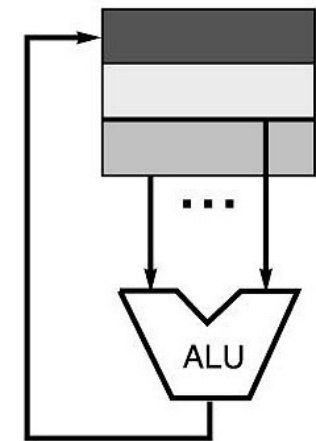
  sub R10, R9, R8   /*   A*B - (A+C*B)   */

R3 = R1 +,-,*,/ R2

Load-Store: Pros and Cons

- pros
  - ✓ simple, fixed length instruction encodings
  - ✓ instructions take similar number of cycles
  - ✓ relatively easy to pipeline and make superscalar

fancy techniques to improve the performance

- cons
  - ✓ higher instruction count
  - ✓ not all instructions need three operands
  - ✓ dependent on good compiler

Load/Store Architectures

can do:

add r1=r2+r3

load r3, m(address)

store r1, m(address)

forces heavy dependence on registers, which works for today's cpus

cannot do

add r1=r2+m(address)

more instructions (-)

fast implementation (e.g., easy pipelining) (+)

easier to keep instruction lengths fixed (-)

Registers: Advantages and Disadvantages

Additional explanation on the previous slides

- **advantages**
  - ✓ faster than cache or main memory (no addressing mode or tags)
  - ✓ deterministic (no misses)
  - ✓ can replicate (multiple read ports)
  - ✓ short identifier (typically 3 to 8 bits)
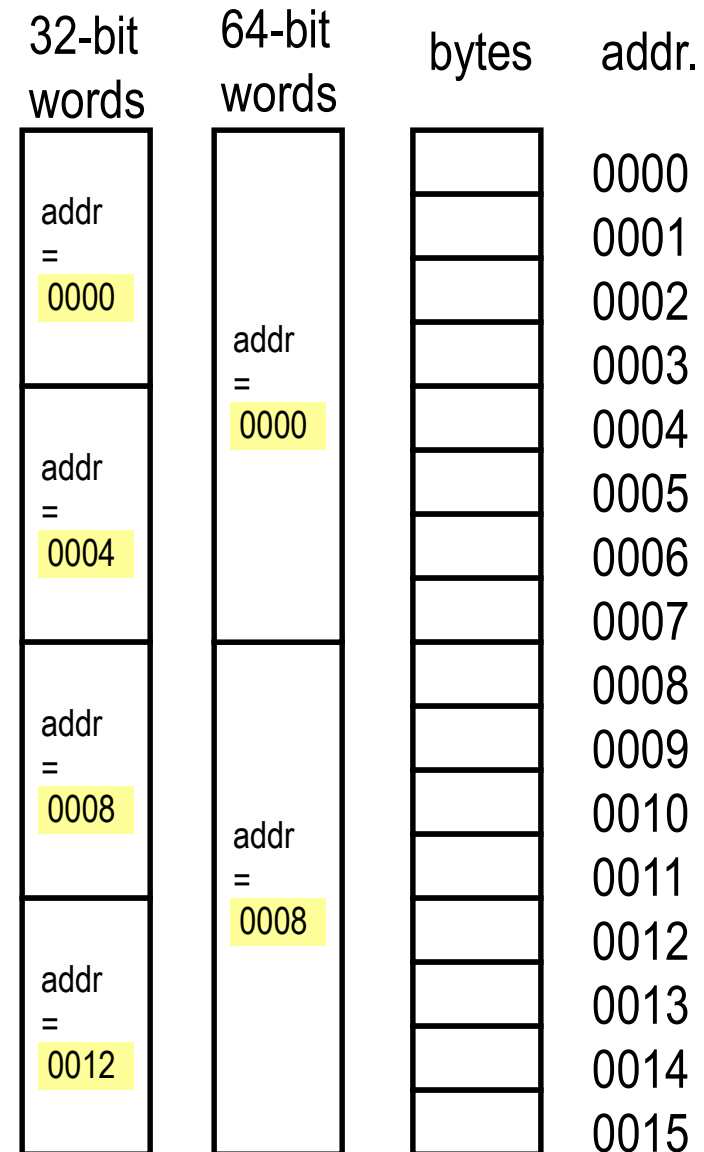 - ✓ reduce memory traffic

- **disadvantages**
  - ✓ need to save and restore on procedure calls and context switch
  - ✓ can't take the address of a register (for pointers)
  - ✓ fixed size (can't store strings or structures efficiently)
  - ✓ compiler must manage
  - ✓ limited number

every ISA designed after 1980 uses a load-store ISA (i.e RISC, to simplify CPU design).

Word-Oriented Memory Organization

- memory is byte addressed and provides access for bytes (8 bits), half words (16 bits), words (32 bits), and double words(64 bits).

- addresses specify byte locations
  - ✓ address of first byte in word
  - ✓ addresses of successive words differ by 4 (32-bit) or 8 (64-bit)

| 32-bit words	64-bit words	bytes	addr.
addr = 0000	addr = 0000		0000
			0001
			0002
			0003
addr = 0004			0004
			0005
			0006
			0007
addr = 0008	addr = 0008		0008
			0009
			0010
			0011
addr = 0012			0012
			0013
			0014
			0015

Byte Ordering

- how should bytes w/i multi-byte word be ordered in memory?

- conventions
  - ✓ Sun's, Mac's are "big endian" machines
    - o least significant byte has highest address
  - ✓ Alphas, PC's are "little endian" machines
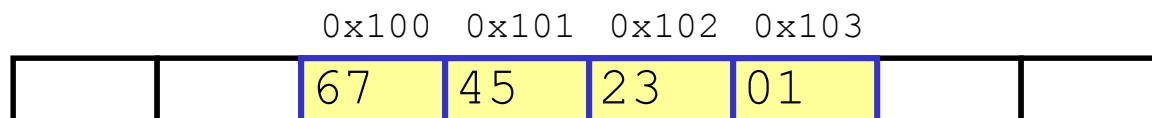   - o least significant byte has lowest address

Byte Ordering Example

- **big endian**
  - ✓ least significant byte has highest address

- **little endian**
  - ✓ least significant byte has lowest address

- **example**
  - ✓ variable x has 4-byte representation 0x01234567
  - ✓ address given by &x is 0x100

big endian

		0x100	0x101	0x102	0x103		
		01	23	45	67		

little endian

		0x100	0x101	0x102	0x103		
		67	45	23	01		
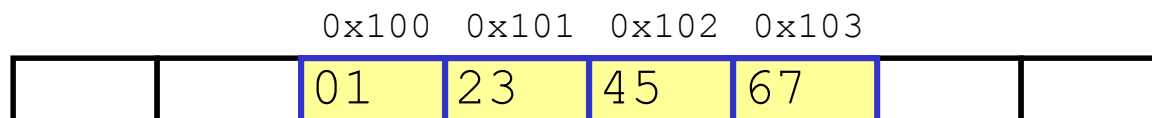
ECE 411 COMPUTER ORGANIZATION AND DESIGN

Reading Byte-Reversed Listings

- disassembly
  - ✓ text representation of binary machine code
  - ✓ generated by program that reads the machine code

- example fragment

```
address              instruction code                 assembly rendition
8048365:    5b                                pop     %ebx
8048366:    81 c3 ab 12 00 00                 add     $0x12ab,%ebx
804836c:    83 bb 28 00 00 00 00              cmpl    $0x0,0x28(%ebx)
```

- deciphering numbers
  - ✓ value: 0x12ab
  - ✓ pad to 4 bytes: 0x000012ab
  - ✓ split into bytes: 00 00 12 ab
  - ✓ reverse: ab 12 00 00

Types of Addressing Modes (VAX)

addressing mode	example	action
1. register direct	add R4, R3	R4 <- R4 + R3
2. immediate	add R4, #3	R4 <- R4 + 3
3. displacement	add R4, 100(R1)	R4 <- R4 + M[100 + R1]
4. register indirect	add R4, (R1)	R4 <- R4 + M[R1]
5. indexed	add R4, (R1 + R2)	R4 <- R4 + M[R1 + R2]
6. direct	add R4, (1000)	R4 <- R4 + M[1000]
7. memory indirect	add R4, @(R3)	R4 <- R4 + M[M[R3]]
8. autoincrement	add R4, (R2)+	R4 <- R4 + M[R2]
		R2 <- R2 + d
9. autodecrement	add R4, (R2)-	R4 <- R4 + M[R2]
		R2 <- R2 - d
10. scaled	add R4, 100(R2)[R3]	R4 <- R4 + M[100 + R2 + R3*d]

Studies by [Clark and Emer] indicate that modes 1-4 account for 93% of all operands on the VAX.

Types of Operations

- arithmetic and logic: AND, ADD

- data transfer: MOVE, LOAD, STORE

- control   BRANCH, JUMP, CALL

- system   OS CALL, VM

- floating point   ADDF, MULF, DIVF

- decimal   ADDD, CONVERT

- string   MOVE, COMPARE

- graphics   (DE)COMPRESS

80x86 Instruction Frequency

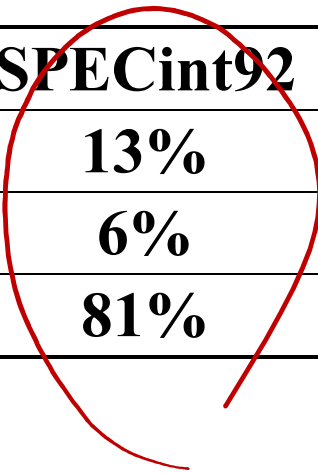| Rank	Instruction	Frequency	
1	load	22%	→ 20%
2	branch	20%	
3	compare	16%	
4	store	12%	
5	add	8%	
6	and	6%	
7	sub	5%	
8	register move	4%	
9	call	1%	2x
10	return	1%	
Total		96%	

Relative Frequency of Control Instructions

- design hardware to handle branches quickly, since these occur most frequently

Operation	SPECint92	SPECfp92
Call/Return	13%	11%
Jumps	6%	4%
Branches	81%	87%

Instruction formats

what does each bit mean?

- machine needs to determine quickly,
  - ✓ "this is a 6-byte instruction"
  - ✓ "bits 7-11 specify a register"
  - ✓ ...
  - ✓ serial decoding bad

- having many different instruction formats...
  - ✓ complicates decoding
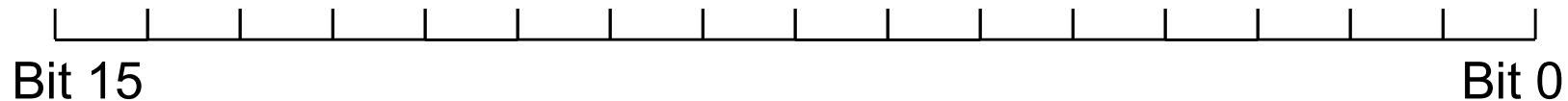 - ✓ uses instruction bits (to specify the format)

what would be a good thing about having many different instruction formats?
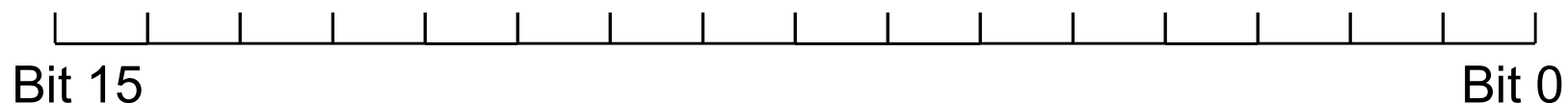
LC-3b Instruction Formats

Look at the slides from home.

- ADD, AND (w/o immediate)

Opcode	DR	SR1	0	00	SR2

Bit 15   Bit 0

- ADD, AND (w/ immediate), NOT

Opcode	DR	SR1	1	Imm5

Bit 15   Bit 0

ECE 411 COMPUTER ORGANIZATION AND DESIGN

MIPS Instruction Formats

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
r format	OP	rs	rt	rd	sa	funct
i format	OP	rs	rt	immediate		
j format	OP	target				

- for instance, "add r1, r2, r3" has
  - ✓ OP=0, rs=2, rt=3, rd=1, sa=0 (shift amount), funct=32
  - ✓ 000000, 00010, 00011, 00001, 00000, 100000

- opcode (OP) tells the machine which format

if you want to know more about MIPS instruction set, please refer to:
https://en.wikipedia.org/wiki/MIPS_instruction_set

MIPS ISA Tradeoffs

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
r format	OP	rs	rt	rd	sa	funct
i format	OP	rs	rt	immediate		
j format	OP	target				

- what if?
  - ✓ 64 registers
  - ✓ 20-bit immediate
  - ✓ 4 operand instruction (e.g., Y = AX + B)

think about how sparsely the bits are used

Conditional branch

- how do you specify the destination of a branch/jump?
  - ✓ theoretically, the destination is a full address
    - ○ 16 bits for LC3b
    - ○ 32 bits for MIPS

- studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else)
  - ✓ we can specify a relative address in much fewer bits than an absolute address
  - ✓ e.g., beq $1, $2, 100=>if ($1 == $2) PC = PC + 100 * 4

- how do we specify the condition of the branch?

MIPS conditional branches

- beq, bne, beq r1, r2, addr=>if (r1 == r2) goto addr

- slt $1, $2, $3 => if ($2 < $3) $1 = 1; else $1 = 0

- these, combined w/ $0, can implement all fundamental branch conditions
  - ✓ always, never, !=, = =, >, <=, >=, <, >(unsigned), <= (unsigned), ...

if (i<j)
    w = w+1;
else
    w = 5;
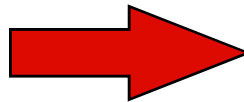
```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```

Jumps

- need to be able to jump to an absolute address sometimes
  - ✓ jump -- j 10000 => PC = 10000

- need to be able to do procedure calls and returns
  - ✓ jump and link--jal 100000 => $31 = PC + 4; PC = 10000
    - o used for procedure calls
  - ✓ jump register -- jr $31 => PC = $31
    - o used for returns, but can be useful for lots of other things

OP	target

ECE 411 COMPUTER ORGANIZATION AND DESIGN

Announcement

- next lecture
  - ✓ performance, energy, and power metric
- MP assignment
  - ✓ MP0 due on 1/23