EE 204 Computer Architecture

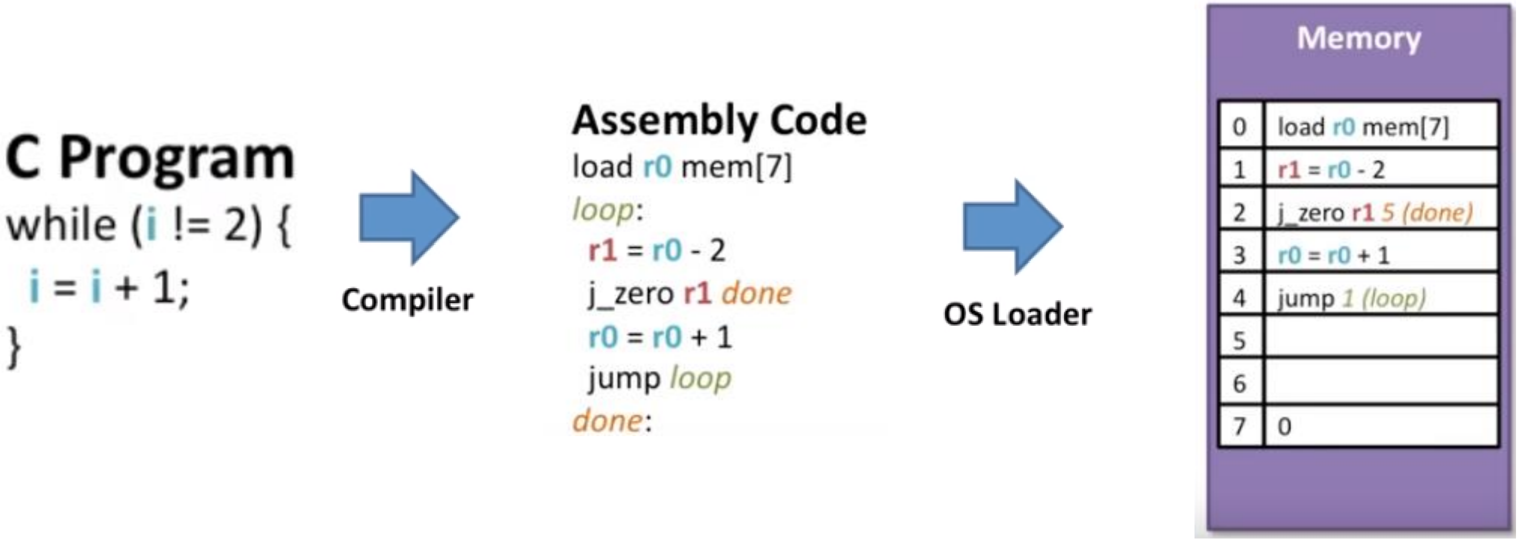
Instruction Set Architecture: An Introduction

Instructor: Dr. Hassan Jamil Syed

Courtesy of Prof. Yifeng Zhu @ U. of Maine and Hong Jiang and Prof. David Black-Schaffer @ Uppsala University

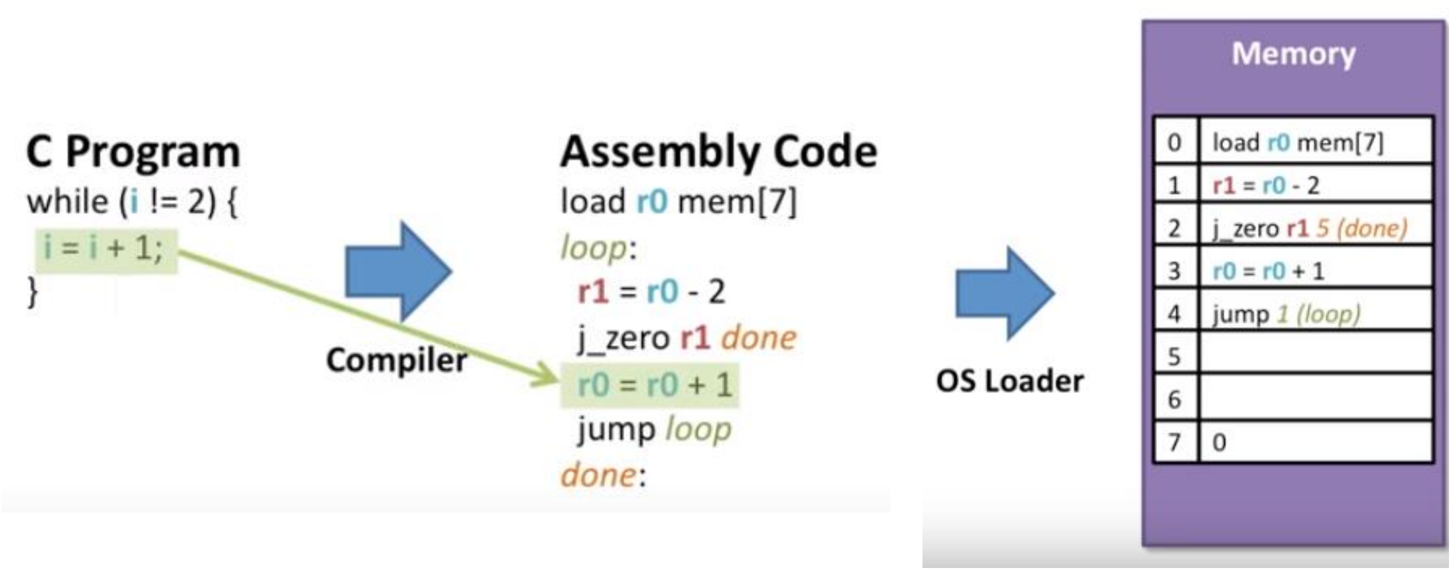
Fall 2019

Programming a processor

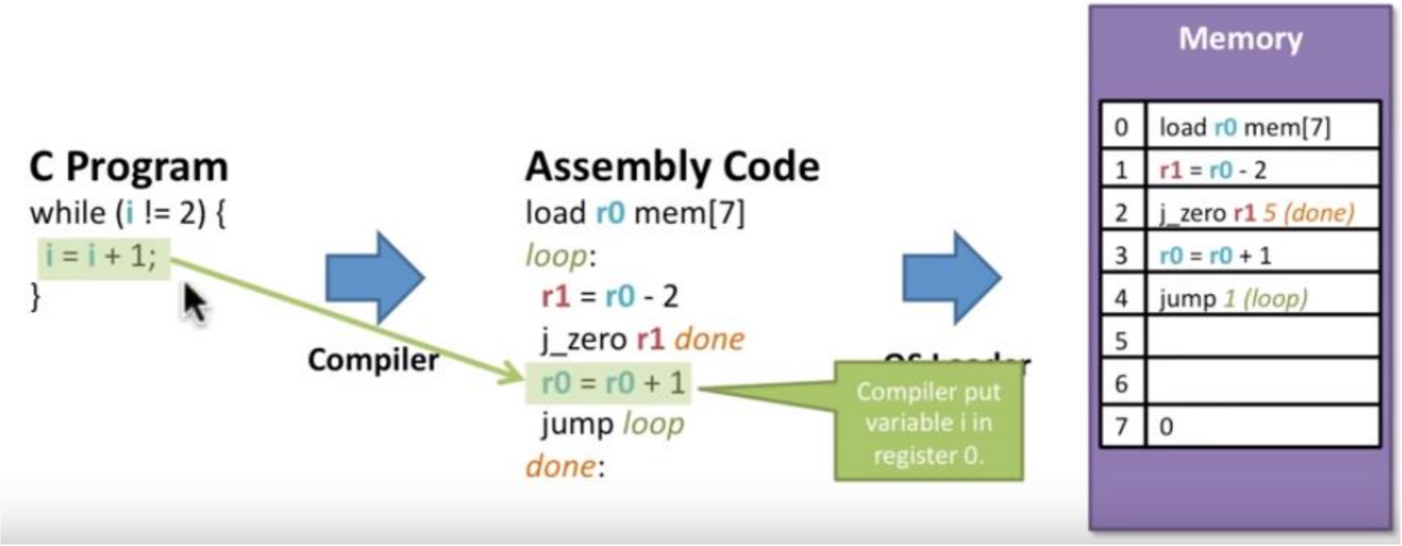


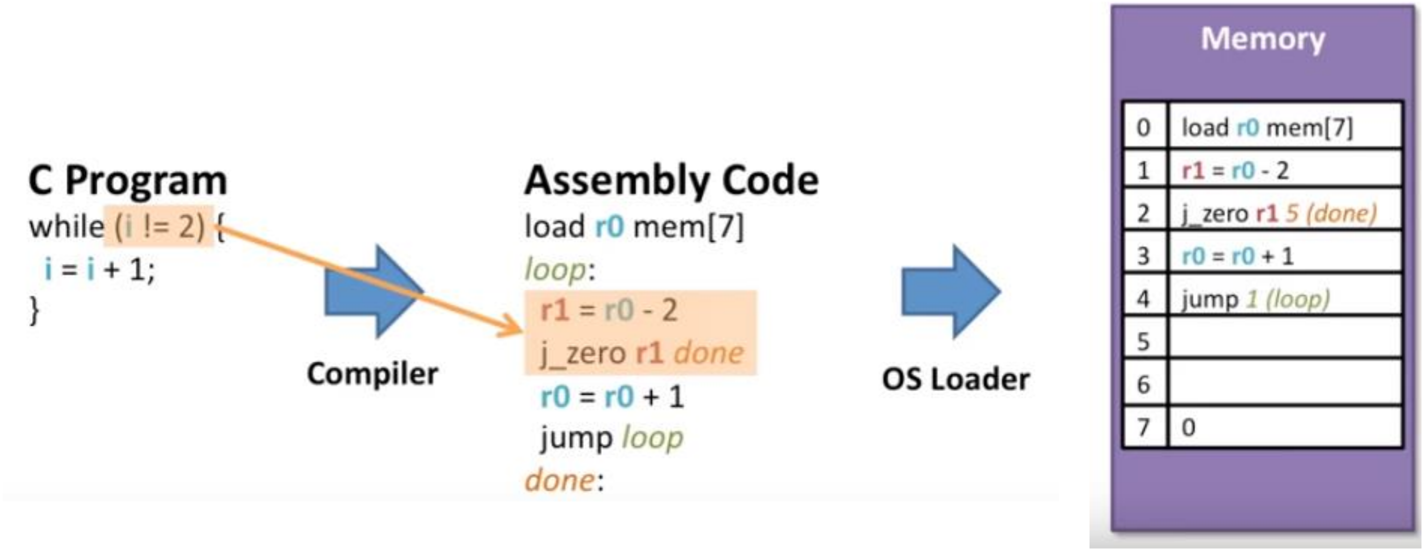
Source: Introduction to computer architecture by David Black-Schaffer
https://www.youtube.com/watch?v=PlavjNH_RRU&t=6s

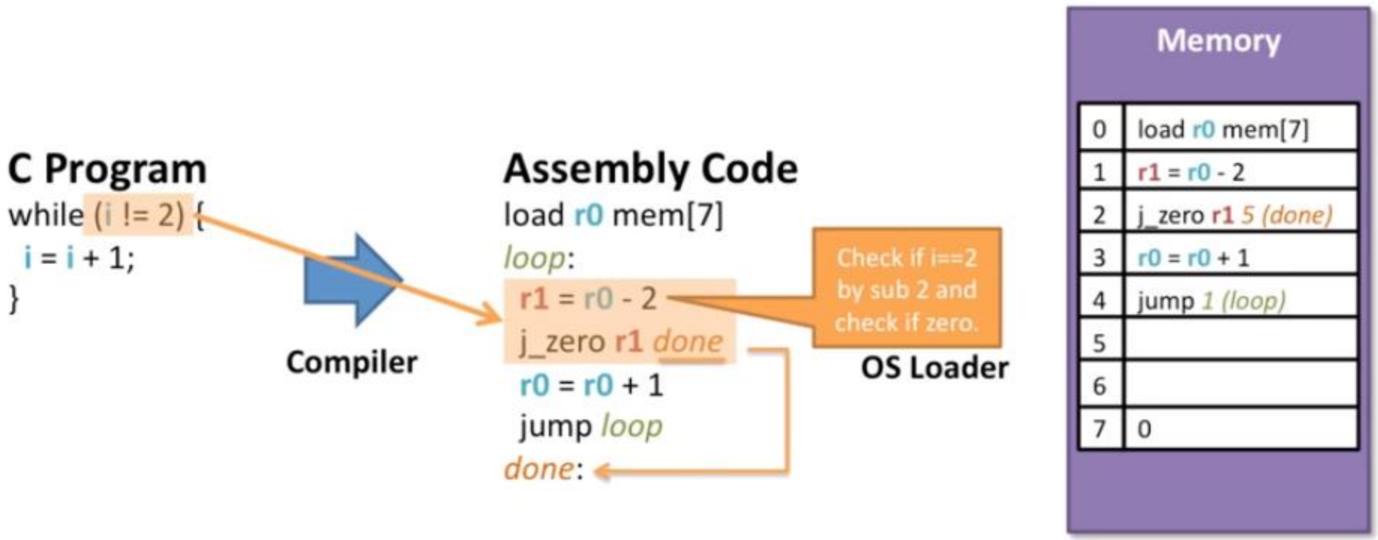
Programming a processor



Programming a processor



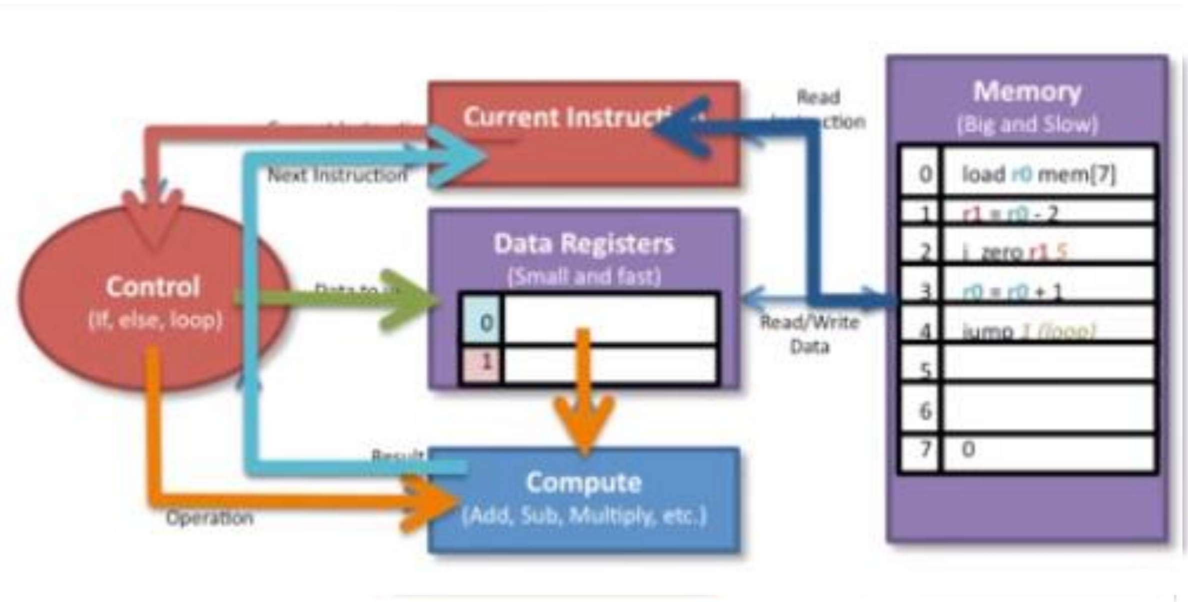

C Program
while (i != 2) {
i = i + 1;
}
Compiler
Assembly Code
load r0 mem[7]
loop:
r1 = r0 - 2
j_zero r1 done
r0 = r0 + 1
jump loop
done:
OS Loader
Memory
0 load r0 mem[7]
1 r1 = r0 - 2
2 j_zero r1 5 (done)
3 r0 = r0 + 1
4 jump 1 (loop)
5
6
7 0


C Program
while (i != 2) {
i = i + 1;
}
Compiler
Assembly Code
load r0 mem[7]
loop:
r1 = r0 - 2
j_zero r1 done
r0 = r0 + 1
jump loop
done:
Check if i==2 by sub 2 and check if zero.
OS Loader
Memory
0 load r0 mem[7]
1 r1 = r0 - 2
2 j_zero r1 5 (done)
3 r0 = r0 + 1
4 jump 1 (loop)
5
6
7 0

Walking through program execution

- **What will processor do?**
  - **1. load the instruction.**
  - **2. Figure out what operation to do.**
  - **3. Figure out what data to use.**
  - **4. Do the computation.**
  - **5. Figure out the next instruction.**

Repeat the process over and over again.



Outline

- **Instruction Set Overview**
  - **Classifying Instruction Set Architectures (ISAs)** ⇦
  - **Memory Addressing**
  - **Types of Instructions**
- **MIPS Instruction Set (Topic of next lecture)**

Instruction Set Architecture (ISA)

- **Serves as an interface between 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

software

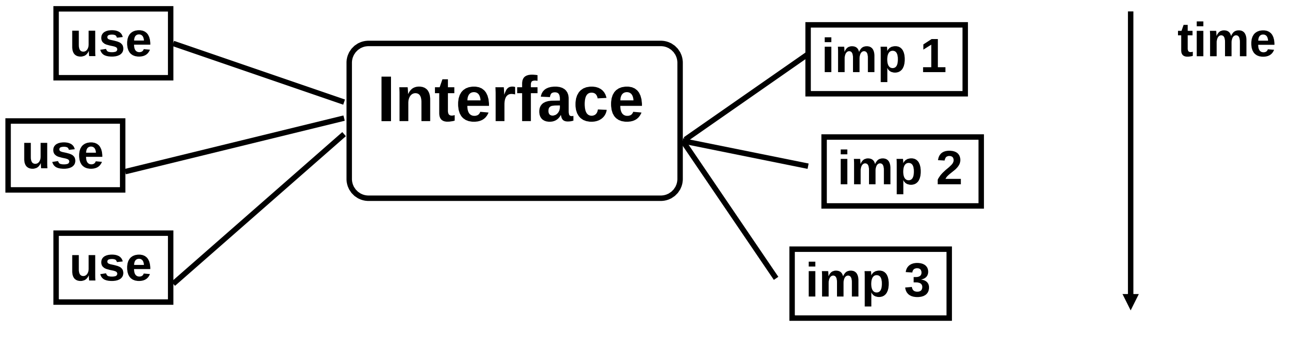
instruction set

hardware

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**



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 . . .**

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)

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]

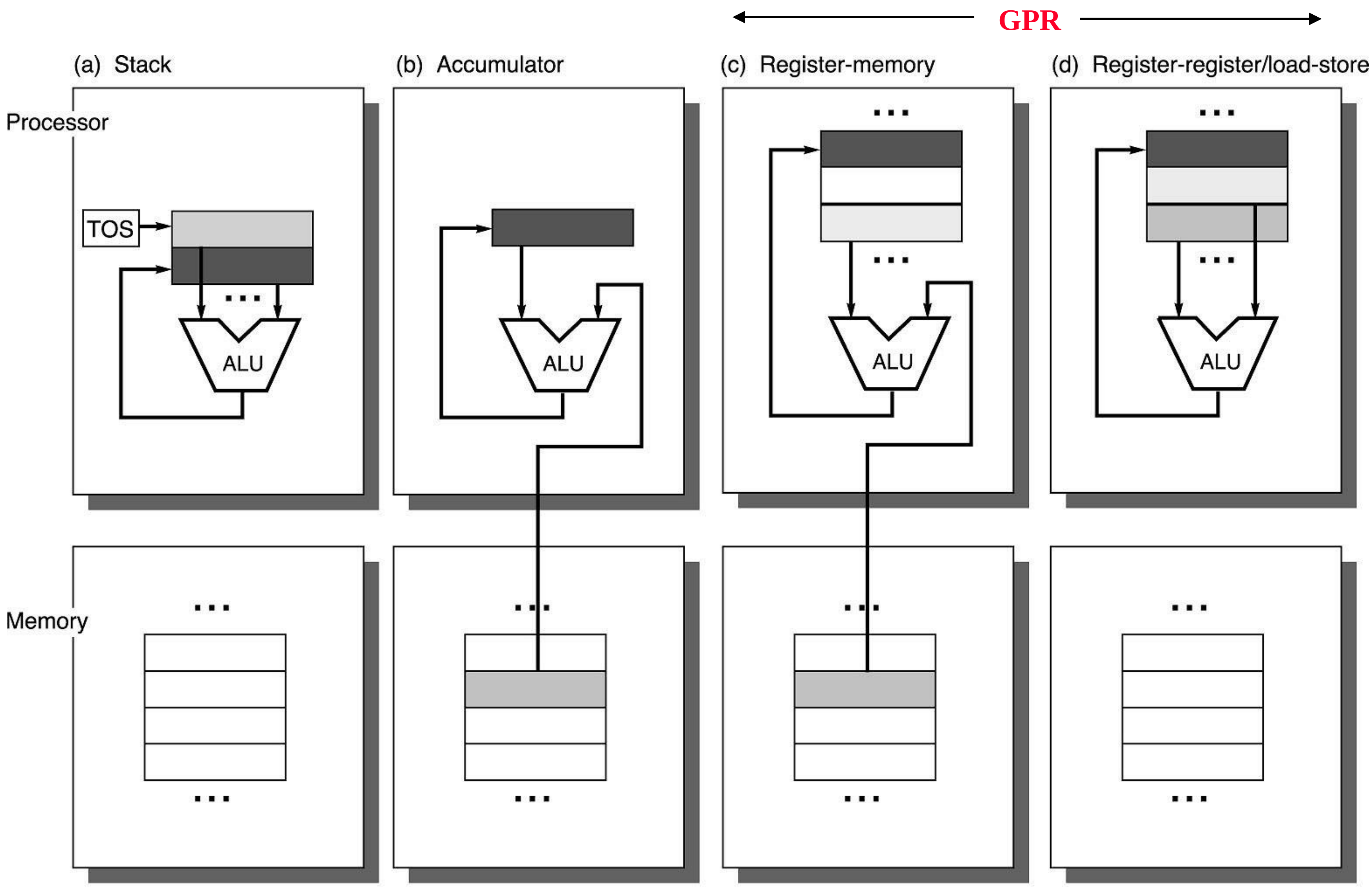
Register-Memory (1970s to present, e.g. 80x86):

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):

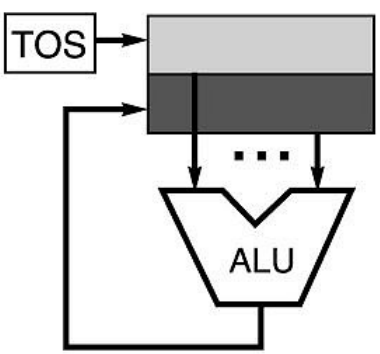
3-address	add R1, R2, R3	R1 ← R2 + R3
	load R1, X(R2)	R1 ← mem[R2]
	store R1, C	mem[R1] ← R2

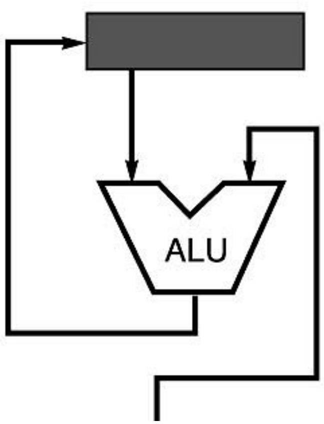
Operand Locations in Four ISA Classes



Code Sequence C = A + B for Four Instruction Sets

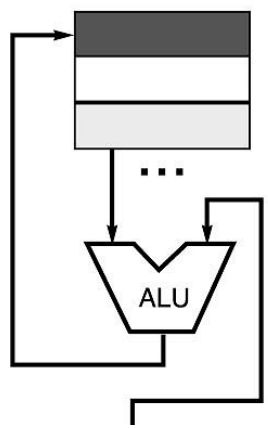
Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1,A Load R2, B Add R3, R1, R2 Store C, R3





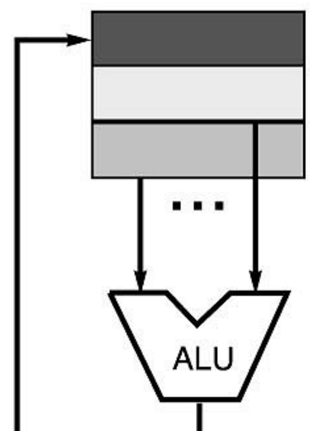
memory

acc = acc + mem[C]



memory

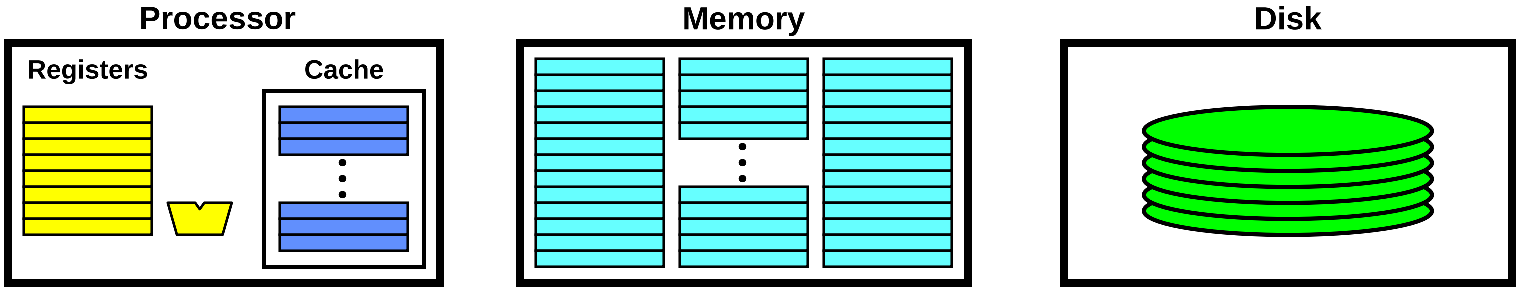
R1 = R1 + mem[C]



R3 = R1 + R2

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)

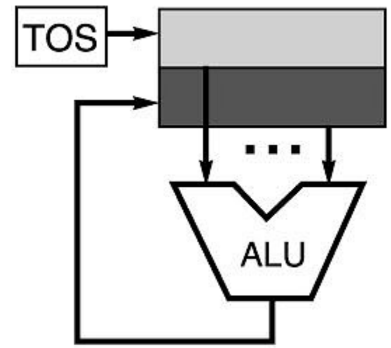


Stack Architectures

- **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

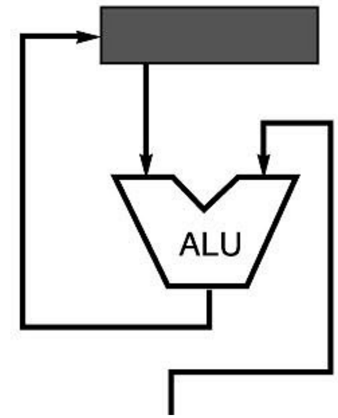
A	B	A*B	A	C	B	B*C	A+B*C	result
	A		A*B	A	C	A	A*B	
				A*B	A	A*B		
					A*B			

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

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

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

  load B

  mul C

  add A

  store D

  load A

  mul B

  sub D

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

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)

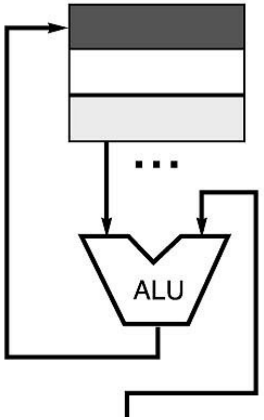
- **Cons**
  - Very high memory traffic (especially if 3 operands)
  - Variable number of clocks per instruction
  - With two operands, more data movements are required

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 without 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

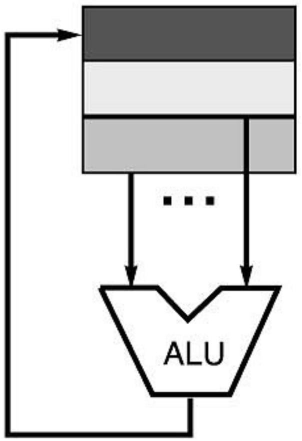
Load-Store Architectures

- **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
- **Cons**
  - Higher instruction count
  - Not all instructions need three operands
  - Dependent on good compiler

Registers: Advantages and Disadvantages

- 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

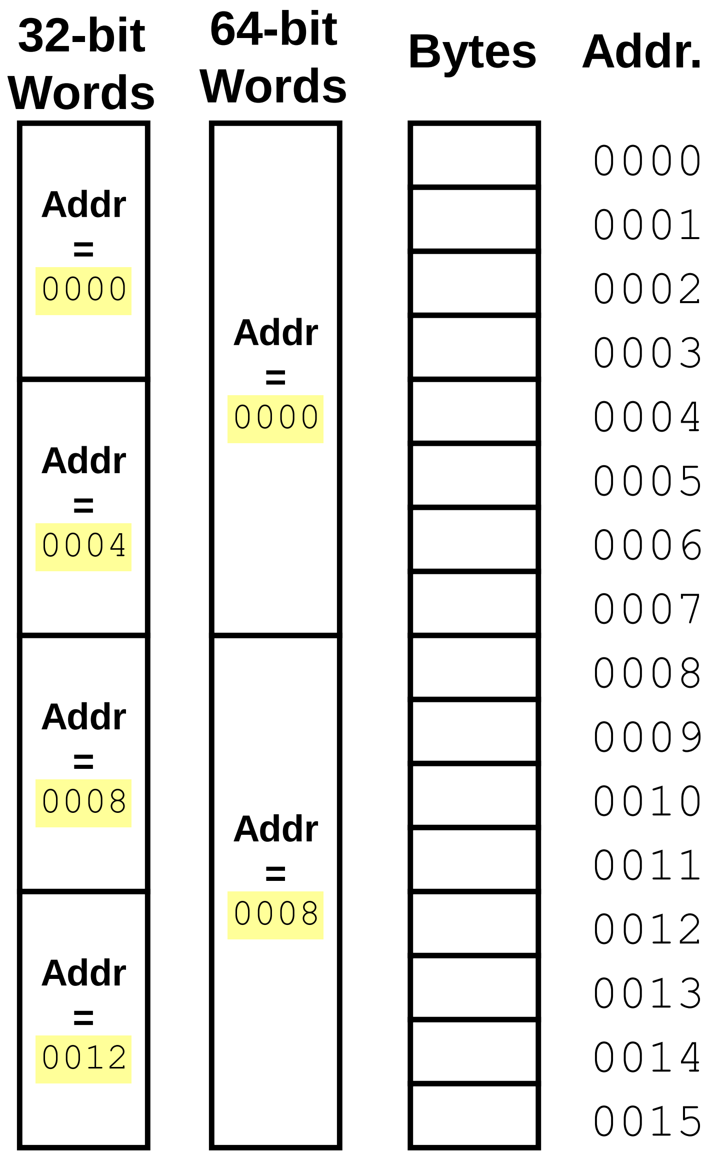
- Disadvantages
  - Need to save and restore on procedure calls and context switch
  - Can't take the address of a register (for pointers)
  - Fixed size (can't store strings or structures efficiently)
  - Compiler must manage
  - Limited number

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)



Byte Ordering

- **How should bytes within multi-byte word be ordered in memory?**
- **Conventions**
  - **Sun's, Mac's are "Big Endian" machines**
    - » **Least significant byte has highest address**
  - **Alphas, PC's are "Little Endian" machines**
    - » **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		

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
• Floating Point	ADDF, MULF, DIVF
• Decimal	ADDD, CONVERT
• String	MOVE, COMPARE

80x86 Instruction Frequency

Rank	*Instruction*	*Frequency*
1	load	22%
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%
10	return	1%
Total		96%

Relative Frequency of Control Instructions

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

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

Summery

- **Instruction Set Overview**
  - Classifying Instruction Set Architectures (ISAs)
  - Memory Addressing
  - Types of Instructions
- **MIPS Instruction Set (Topic of next class)** ⇦
  - Overview
  - Registers and Memory
  - Instructions