CMPE 411
Computer Architecture

Lecture 3

Addressing Mode & Architectural Design Guidelines

September 7, 2017

www.csee.umbc.edu/~younis/CMPE411/
CMPE411.htm

Lecture's Overview

❑ *Previous Lecture:*

- Instruction set architecture and CPU operations
  (The stored-program concept)

- Instruction types, operands and operations
  (R-type and I-type MIPS instructions format)

- Decision making and repetition of instruction execution
  (*bne*, *beq* and *j* instructions)

- Supporting procedure and context switching
  (Stack operations, nested procedure call, *jal* and *jr* instructions)

❑ *This Lecture:*

- Other styles of MIPS addressing

- Program starting steps

- Architectural design guidelines

MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement

  add $t0, $s1, $s2

  sub $t0, $s1, $s2

- Each arithmetic instruction performs only one operation
- Each arithmetic instruction fits in 32 bits and specifies exactly three operands

  destination ← source1   op   source2

- Operand order is fixed (destination first)
- Those operands are all contained in the datapath's register file ($t0,$s1,$s2) – indicated by $

MIPS Memory Access Instructions

- MIPS has two basic data transfer instructions for accessing memory

  lw $t0, 4($s3) #load word from memory

  sw $t0, 8($s3) #store word to memory

- The data is loaded into (lw) or stored from (sw) a register in the register file – a 5 bit address

- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value

  – A 16-bit field meaning access is limited to memory locations within a region of $\pm 2^{13}$ or 8,192 words ($\pm 2^{15}$ or 32,768 bytes) of the address in the base register

  – Note that the offset can be positive or negative

MIPS Instruction format

❑ Register-format instructions:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

op: Basic operation of the instruction, traditionally called opcode

rs: The first register source operand

rt: The second register source operand

rd: The register destination operand, it gets the result of the operation

shmat: Shift amount (explained in future lectures)

funct: This field selects the specific variant of the operation of the op field

❑ Immediate-type instructions:

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

- Some instructions need longer fields than provided for large value constant
- The 16-bit address means a load word instruction can load a word within a region of $\pm 2^{15}$ bytes of the address in the base register
- Example: lw $t0, 32($s3)   # Temporary register $t0 gets A[8]

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32	N/A
sub	R	0	reg	reg	reg	0	34	N/A
lw	I	35	reg	reg	N/A	N/A	N/A	address
sw	I	43	reg	reg	N/A	N/A	N/A	address

Constant or Immediate Operands

❑ Use of constants is common in programs, e.g. incrementing an index of an array, counting loop iterations, etc.

❑ In the C compiler "gcc", 52% of arithmetic operands involve constants while in the circuit simulation program "spice" it is 69%

❑ Inefficient support:

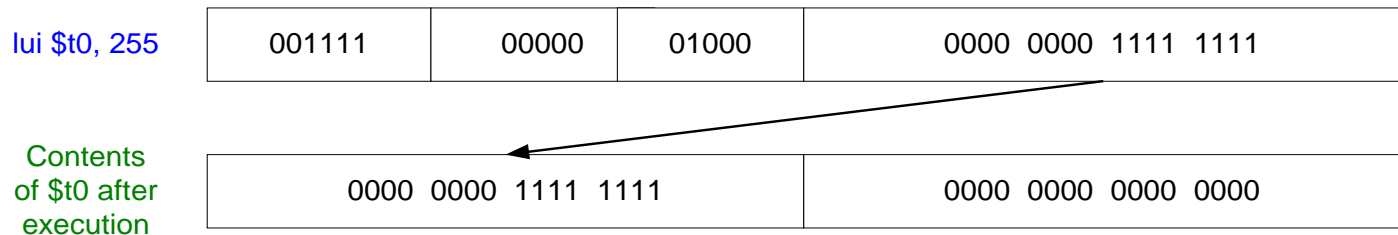Use memory address (AddrConstant4) to reference a stored constant
lw $t0, AddrConstant4($zero)   # $t0 = constant 4
add $sp, $sp, $t0   # $sp = $sp + 4

❑ MIPS handles 16-bit constant efficiently by including the constant value in the address field of an I-type instruction (Immediate-type)

addi $sp, $sp, 4   #$sp = $sp + 4

❑ For large constants that need more than 16 bits, a load upper-immediate (*lui*) instruction is used to concatenate the second part

| lui $t0, 255 | 001111 | 00000 | 01000 | 0000 0000 1111 1111 |

| Contents of $t0 after execution | 0000 0000 1111 1111 | 0000 0000 0000 0000 |

❑ The compiler or the assembler break large constants into 2 pieces and reassemble them in a register

Addressing in Branches & Jumps

❑ I-type instructions leaves only 16 bits for address reference limiting the size of the jump

❑ MIPS branch instructions use the address as an increment to the PC allowing the program to be as large as 2^{32} (called *PC-relative addressing*)

❑ Since the program counter gets incremented prior to instruction execution, the branch address is actually relative to (PC + 4)

❑ MIPS also supports an J-type instruction format for large jump instructions

op	address
6 bits	26 bits

❑ The 26-bit address in a J-type instruct. is concatenated to upper 8 bits of PC

Label	Instruction		Addr						
Loop:	**add**	**$t1, $s3, $s3**	80000	0	19	19	9	0	32
	add	**$t1, $t1, $t1**	80004	0	9	9	9	0	32
	add	**$t1, $t1, $s6**	80008	0	9	22	9	0	32
	lw	**$t0, 0($t1)**	80012	35	9	8	0		
	bne	**$t0, $s5, Exit**	80016	5	8	21	8		
	add	**$s3, $s3, $s4**	80020	0	19	20	19	0	32
	j	**Loop**	80024	2	80000				
			80028	. . .					
Exit:			80032	35	10	8	15		

Aside: MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

* Slide is courtesy of Mary Jane Irwin

Summary of MIPS Addressing Modes

1. Immediate addressing

op	rs	rt	Immediate

2. Register addressing

op	rs	rt	rd	. . .	funct

Registers

Register

3. Base addressing

op	rs	rt	Address

Register

$+$

Memory

Byte	Halfword	Word

4. PC-relative addressing

op	rs	rt	Address

PC

$+$

Memory

Word
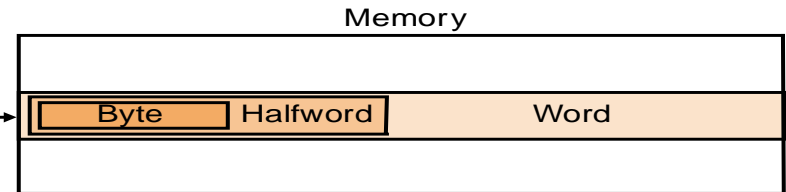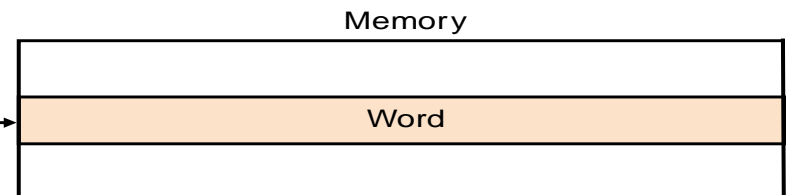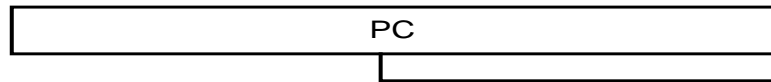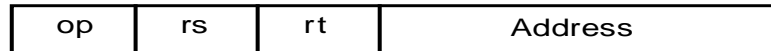
5. Pseudodirect addressing

op	Address

PC

Concatenation

Memory

Word

Starting a Program

C program

↓

Compiler

↓

Assembly language program

↓

Assembler

↓

Object: Machine language module		Object: Library routine (machine language)

↓   ↓

Linker

↓

Executable: Machine language program

↓

Loader

↓
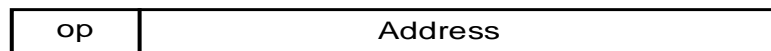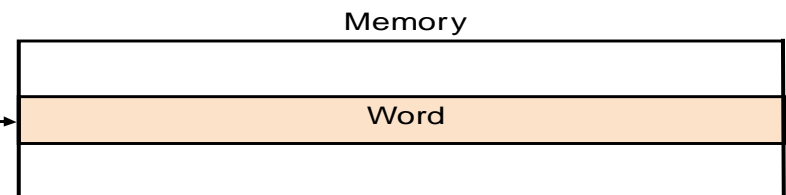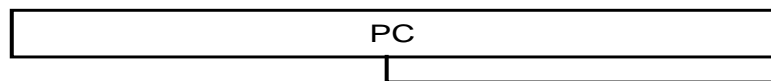
Memory

Object files for Unix typically contains:

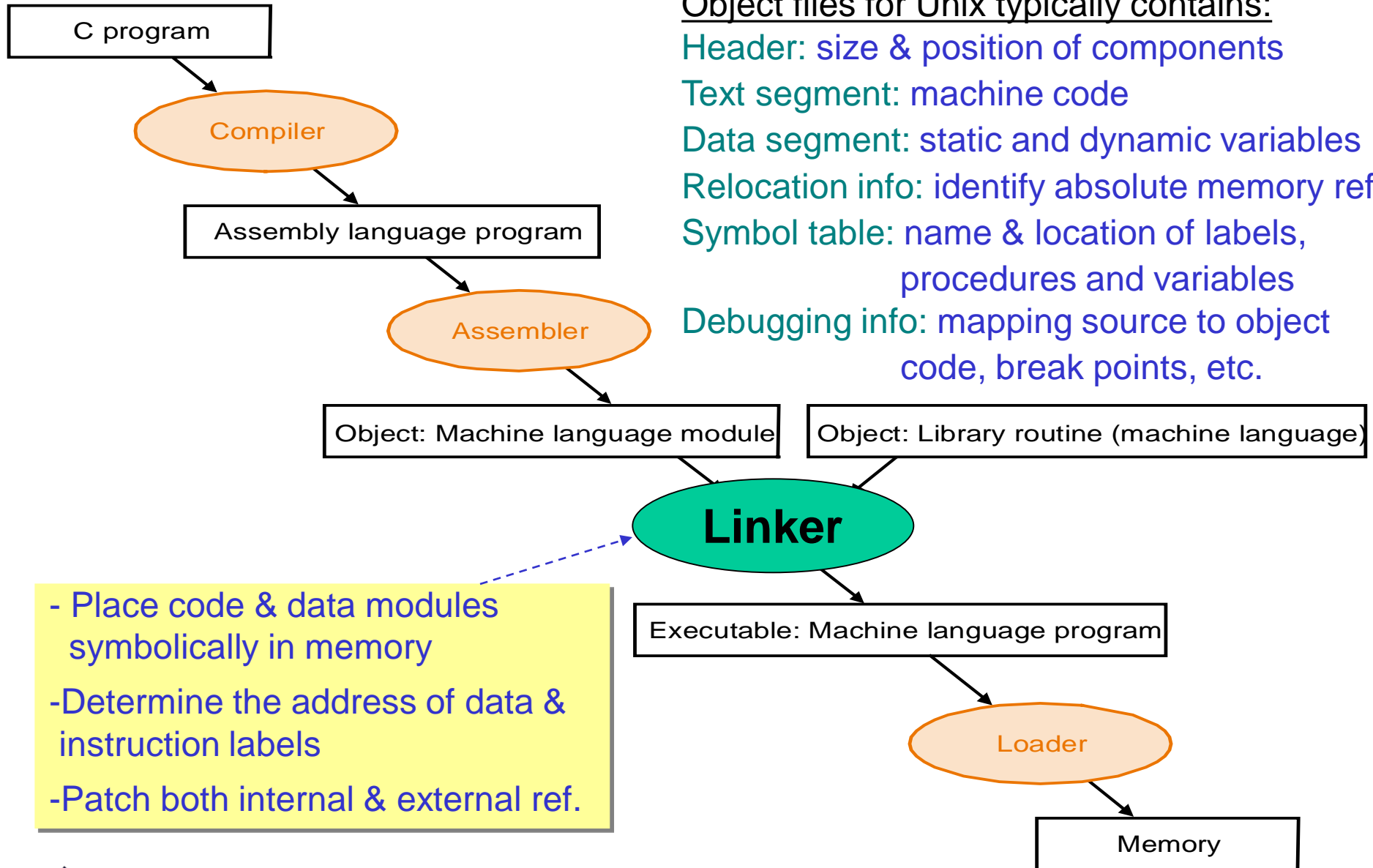Header: size & position of components

Text segment: machine code

Data segment: static and dynamic variables

Relocation info: identify absolute memory ref.

Symbol table: name & location of labels, procedures and variables

Debugging info: mapping source to object code, break points, etc.

- Place code & data modules symbolically in memory

- Determine the address of data & instruction labels

- Patch both internal & external ref.

Linking Object Files

Object file header			
	Name	Procedure A	
	Text size	100$_{hex}$	
	Data size	20$_{hex}$	
Text segment	Address	Instruction	
	0	lw $a0, 0($gp)	
	4	jal 0	
	…	…	
Data segment	0	(X)	
	….	…	
Relocation Info	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	-	
	B	-	

Object file header			
	Name	Procedure B	
	Text size	200$_{hex}$	
	Data size	30$_{hex}$	
Text segment	Address	Instruction	
	0	lw $a0, 0($gp)	
	4	jal 0	
	…	…	
Data segment	0	(Y)	
	….	…	
Relocation Info	Address	Instruction type	Dependency
	0	lw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	-	
	A	-	

Executable file header		
	Text size	300$_{hex}$
	Data size	50$_{hex}$
Text segment	Address	Instruction
	0040 0000$_{hex}$	lw $a0, 8000$_{hex}$($gp)
	0040 0004$_{hex}$	jal 40 0100$_{hex}$
	…	…
	0040 0100$_{hex}$	lw $a1, 8020$_{hex}$($gp)
	0040 0104$_{hex}$	jal 40 0000$_{hex}$
	…	…
Data segment	Address	
	1000 0000$_{hex}$	(X)
	…	…
	1000 0020$_{hex}$	(Y)
	…	…

Assuming the value in $gp is 1000 8000$_{hex}$

Loading Executable Program

$sp ⟶ 7fff ffff   hex

Stack
↓
↑
Dynamic data

$gp ⟶ 1000 8000   hex

Static data

1000 0000   hex

Text

pc ⟶ 0040 0000   hex

Reserved

0
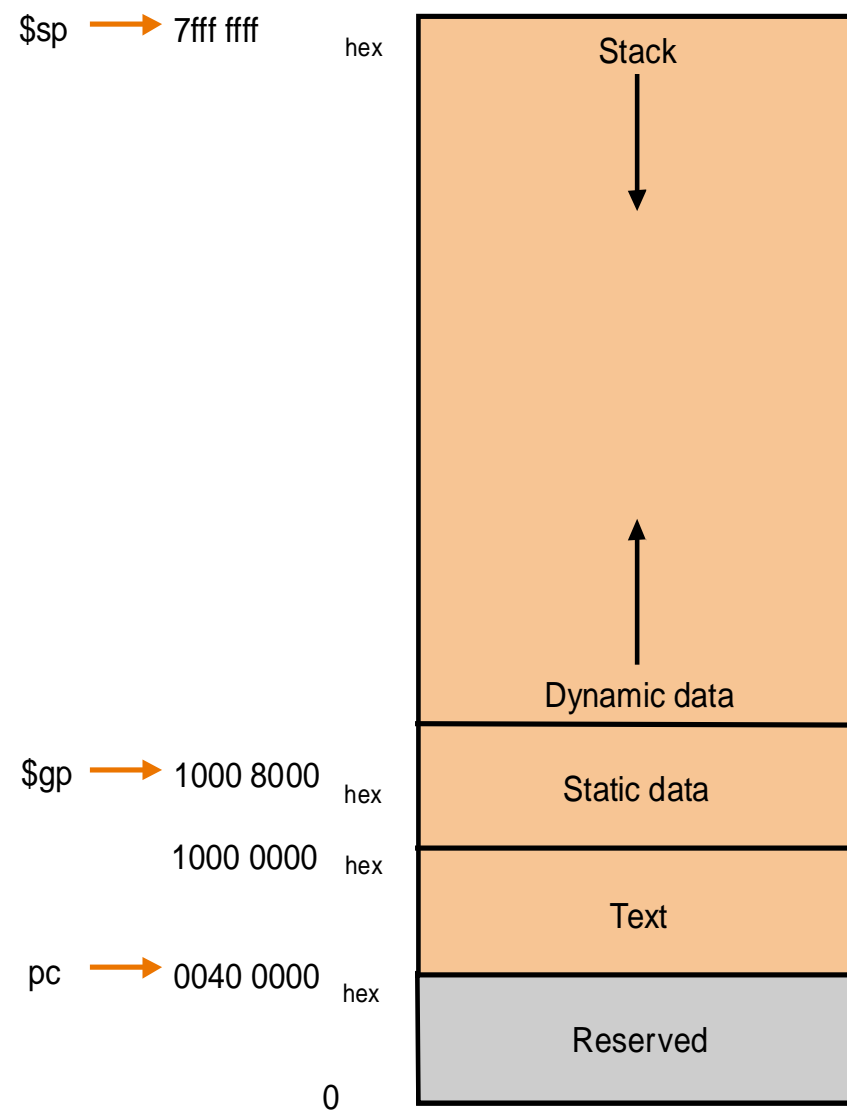
To load an executable, the operating system follows these steps:

❶ Reads the executable file header to determine the size of text and data segments

❷ Creates an address space large enough for the text and data

❸ Copies the instructions and data from the executable file into memory

❹ Copies the parameters (if any) to the main program onto the stack

❺ Initializes the machine registers and sets the stack pointer to the first free location

❻ Jumps to a start-up routines that copies the parameters into the argument registers and calls the main routine of the program

Classifying Instruction Set Architectures

❑ Accumulator Architecture

- Common in early stored-program computers when hardware was so expensive
- Machine has only one register (accumulator) involved in all math. & logical operations
- All operations assume the accumulator as a source operand and a destination for the operation, with the other operand stored in memory

❑ Extended Accumulator Architecture

- Dedicated registers for specific operations, e.g., stack and array index registers, added
- The 8086 microprocessor is a an example of such special-purpose register arch.

❑ General-Purpose Register Architecture

- MIPS is an example of such arch. where registers are not sticking to play a single role
- This type of instruction set can be further divided into:
  - *Register-memory:* allows for one operand to be in memory
  - *Register-register (load-store):* demands all operands to be in registers

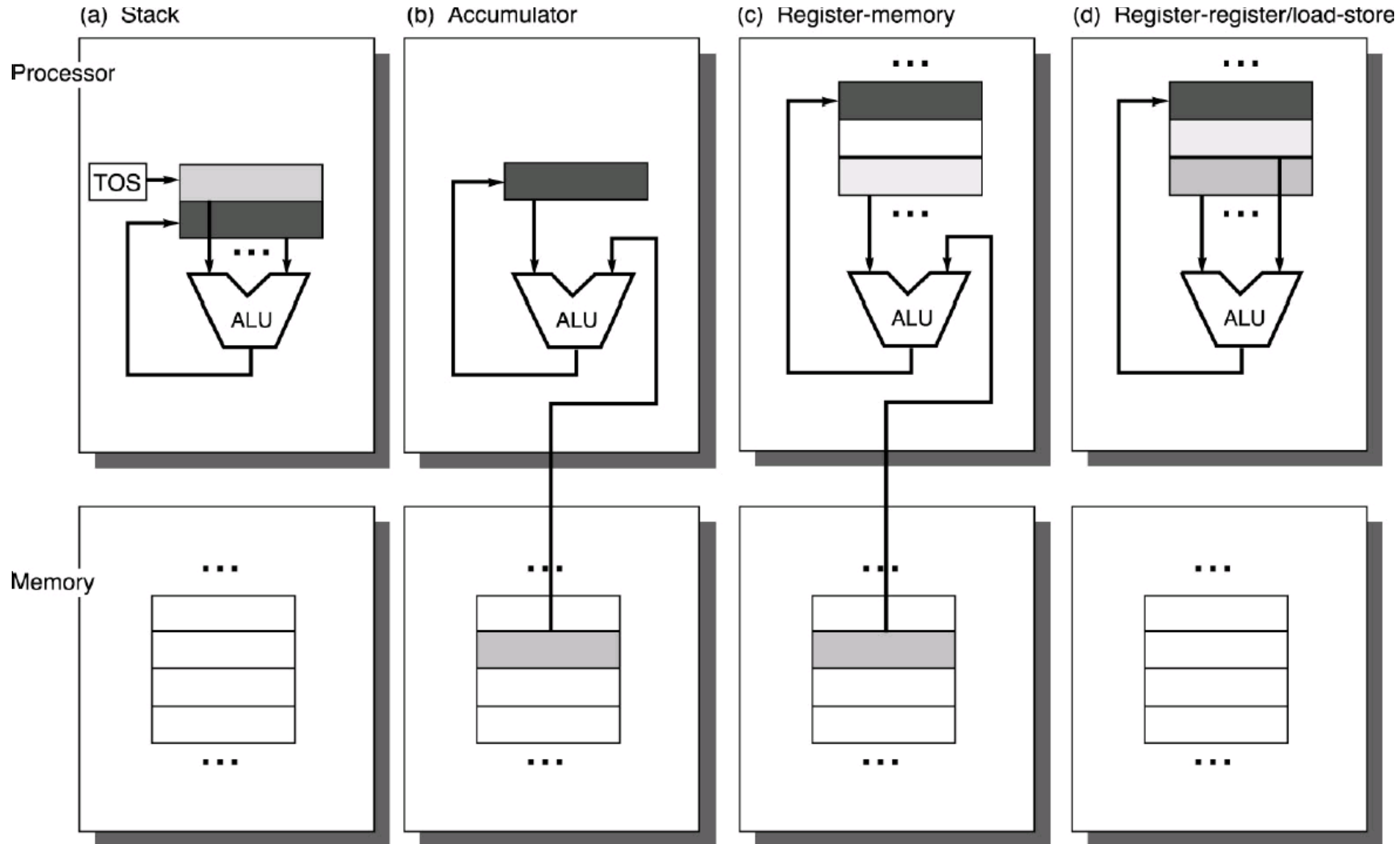| Machine	# general-purpose registers	Architecture style	Year
Motorola 6800	2	Accumulator	1974
DEC VAX	16	Register-memory, memory-memory	1977
Intel 8086	1	Extended accumulator	1978
Motorola 68000	16	Register-memory	1980
Intel 80386	32	Register-memory	1985
PowerPC	32	Load-store	1992
DEC Alpha	32	Load-store	1992

Compact Code and Stack Architectures

❑ When memory is scarce, machines, like Intel 80x86 had variable-length instructions to match varying operand specifications and minimize code size

❑ Stack machines abandoned registers altogether arguing that it is hard for compilers to use them efficiently

❑ Operands are to be pushed on a stack from memory and the results have to be popped from the stack to memory

❑ Operations take their operand by default from the top of the stack and insert the results back onto the stack

❑ Stack machines simplify compilers and lent themselves to a compact instruction encoding

❑ *Example:*   A = B + C

```
Push AddressC        # Top=Top+4; Stack[Top]=Memory[AddressC]
Push AddressB        # Top=Top+4; Stack[Top]=Memory[AddressB]
add                  # Stack[Top-4]=Stack[Top]+Stack[Top-4]; Top=Top-4
Pop AddressA         # Memory[AddressA]=Stack[Top]; Top=Top-4
```

❑ Compact code is important for heralded network computers where programs must be downloaded (e.g. Java-based applications) or space communications

Famous ISA

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

Processor

TOS

ALU

Memory

Other types of Architecture

❑ High-Level-Language Architecture

- In the 1960s, systems software was rarely written in high-level languages and virtually every commercial operating system before Unix was written in assembly
- Some people blamed the code density on the instruction set rather than the programming language
- A machine design philosophy was advocated with the goal of making the hardware more like high-level languages
- The effectiveness of high-level languages, memory size limitation and lack of efficient compilers doomed this philosophy to a historical footnote

❑ Reduced Instruction Set Architecture

- With the recent development in compiler technology and expanded memory sizes less programmers are using assembly level coding
- Instruction set architecture became measurable in the way compilers rather programmable use them
- RISC architecture favors simplifying hardware design over enriching instruction offering relying on compilers to effectively use them to perform complex operations
- Virtually all new architecture since 1982 follows the RISC philosophy of fixed instruction lengths, load-store operations, and limited addressing mode

Principles of Hardware Designs

❶ **Simplicity favors regularity**
- Limited number of register formats
- Fixed number of operands

❷ **Smaller is faster**
- Increased number of registers lengthen signal paths
- Signals take longer when travelling farther → increasing clock cycle time
- It is not an absolute rule, 31 registers may not be faster than 32 registers
- Designers balance the craving of programs for more registers and the desire to keep the clock cycle fast

❸ **Good design demands good compromises**
- Fixed length instructions requires different formats for different instructions' kinds
- Group the operation codes of instructions of similar format to simplify decoding
- Restrict the variation of different formats to limited number of fields and sort them to a predictable order

❹ **Make the common case fast**
- Include program counter based addressing to speed up conditional branching
- Offer immediate addressing for constant operands

Conclusion

❑ *Summary*

➔ Various styles of MIPS addressing

(Register, displacement, immediate, PC-relative and pseudo-direct)

➔ Program starting steps

(Compiler, Assembler, Linker, Loader)

➔ Architectural design guidelines

(keep it *simple, focus on common cases, smart compromises*)

❑ *Next Lecture*

➔ Why measuring computer performance is important

➔ Different performance metrics

➔ Performance comparison

Reading assignment is sections 2.8, 2.9, and 2.15 in textbook