Instructions

18010714 HYO JAE JEON
19012949 TASNAM A D M TANIM
20012763 ASAD AL AZIZ

Instructions

- Computer's language to command the computer hardware.
- Vocabulary of Instructions are "Instruction Set"

Instruction is computer's language to command computer hardwares.
Vocabulary of Instructions are called Instruction Set.

Stored Program Concept

- Designed by John von Neumann.
- Running process
  - The executable file is loaded into RAM.
  - Among the instructions of the loaded executable file, the first-order instruction is fetched to the CPU.
    - The Fetch process proceeds through the import BUS (I/O BUS).
  - Commands are decoded (interpreted) by the Control Unit inside the CPU.
  - The interpreted commands are executed through ALU (Arithmetic Logic Unit).

Stored Program Concept is designed by John von Neumann.

This is running process.

The executable file is loaded into RAM.
Among the instructions of the loaded executable file, the first-order instruction is fetched to the CPU.
The Fetch process proceeds through the import BUS (I/O BUS).
Commands are decoded (interpreted) by the Control Unit inside the CPU.
The interpreted commands are executed through ALU (Arithmetic Logic Unit).

We will show more about this in later.

Representing Instructions

	OPcode	Memory Addressing Mode	Operand
Example	ADD	Register Mode	A, B, C
Meaning	Operation Code	Memory Addressing Mode	Address (Registers)

This is Instruction set style.

The example of Instruction set is Addition of B, C, and store at A.

Op code of Instruction is operation code, in this example, ADD.

Memory Addressing Mode is about use memory address.
In this example, A, B, C is register, so it’s memory addressing mode is register mode.

Operand means Address of memory, or register.

Let’s start with op-code and operand.

OPcode / Operand

- OPcode
  - Operation Code
  - Add, Sub, Load, Store …
- Operand
  - “Address” of memory / Register

An Instruction consists of an operation code, an addressing method, and an operand.

OP code is called operation code.
Example of op codes are add, substract, load, store.. and many other things.

Operand is address of memory, or register.
We can call data from operand, and operation with op-code.

And about Memory Addressing mode.
There are 8 addressing modes exist.

Depending on the computer, it may be separately defined as above or defined in common with the operation method.

These are the addressing modes.

I will introduce modes start Implied Mode.

Implied Mode

- Does not require an address field.
- Implicit operand assignment.

Example	
STORE X	Store AC into Memory[X].
MUL X	Multiply Memory[X] with AC, store at AC.

Implied mode is Memory Address mode that does not requires an address field.
Operands are Implicit.

These are examples of Implied Mode Instructions.

STORE X is store accumulator value into memory address X.

MUL X is multiply, get value from memory address X, and then multply data with accumulator.
Then store at accumulator.

Immediate Mode

- Specify the operand to actually use in the address field.

Example

LDI 100, R1	Load and Initiate ‘100’ value into Register[R1].

Implied mode is Memory Address mode that specify operands to actually use in the address field.

In example, this is load and Initiate Instruction.

It loads and Initiate ‘100’ value into R1, Register.

Direct Addressing Mode

- Directly stores the address of the operand in the address field.
- Advantage
  - Access to memory is done at once.
- Disadvantage
  - The address space of the memory device that can be accessed is limited.

Example

LDA 600

Load Memory[600] into AC.

Direct Addressing Mode is Memory Address mode that directly stores ‘address’ at the operands in the address field.

Advantage of this method is accessing.
Computer can access to memory at one time, so it’s fast.

However, disadvantage of this method is limitation.
Address space of the memory can be accessed is limited.

This is example of Direct Addressing Mode.
LDA 600 is load memory address [600] into accumulator.

Indirect Addressing Mode

- Stores the memory address where the *effective address* is stored in the address field.
- Advantages
  - Accessible memory address space is determined by the length of words that the CPU can access at one time.
- Disadvantage
  - Requires memory accesses two time.

Example

LDA [600]

Load Memory of [Memory[600]] into AC.

If Memory[600] has ‘700’, than ‘LDA 700’

Indirect Addressing Mode is Memory Address mode that stores memory address where effective address is stored in the address field.

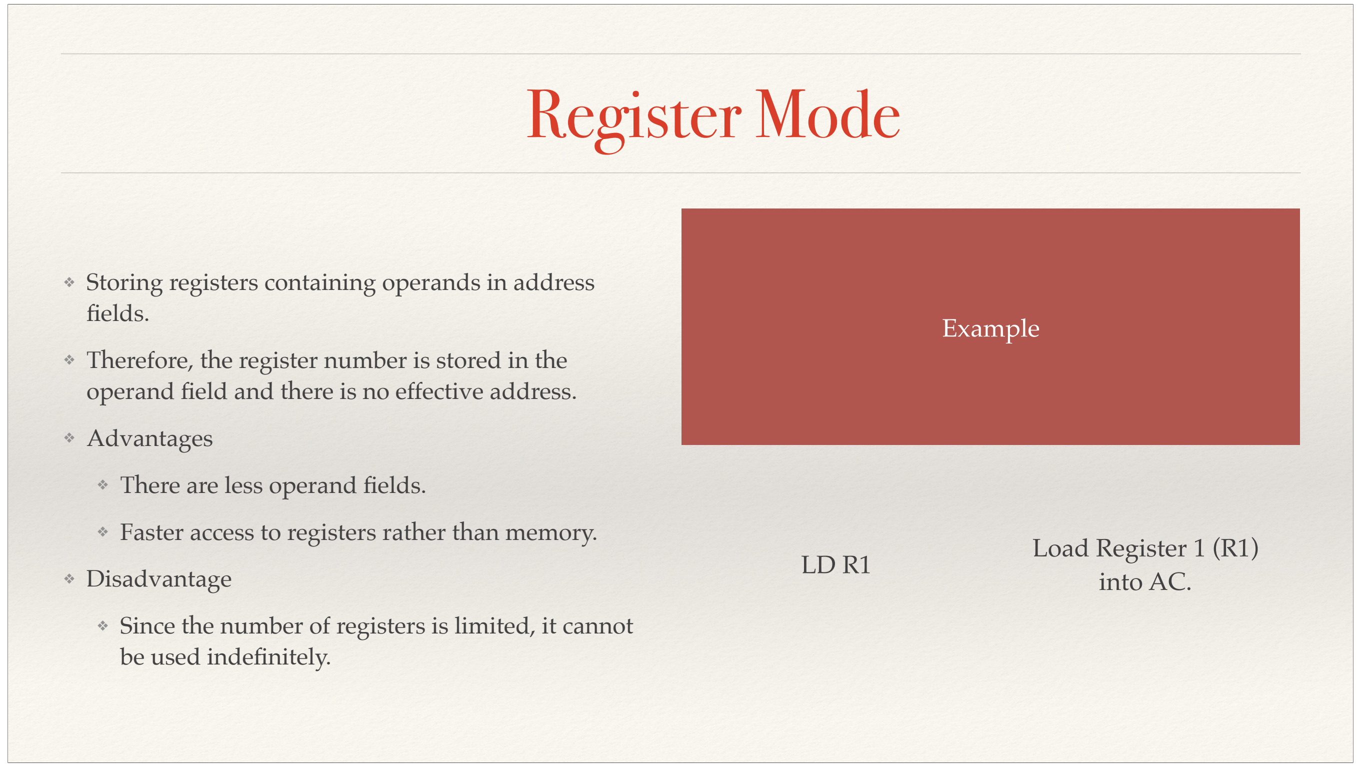
Advantage of this method is length.
Computer can determine access memory address space by the length of words.

However, disadvantage of this method is time.
It requires memory accesses two time.

This is example of Indirect Addressing Mode.

LDA [600], similar as Direct Addressing Mode.

It’s load memory value of memory address 600 into accumulator.
If Memory address 600 has value ‘700’, than it means LDA 700.



Register mode is use ‘registers’ to use data.
So we can use registers in the operands and don’t need to use effective address.

Advantages of register mode is shorter, faster use.
Operand is shorter than effective address modes, and its faster than memory address.

However, registers are limited, so we cannot use infinitly.

Example of register mode is LD R1.
It means Load register data (R1) into accumulator.

But,

Register Indirect Mode

- Designating a register with the memory address value in which the operand is stored in the address field.
- Therefore, in this method, the effective address is the address in the designated register.

Example

LDA (R1)	Load Register 1 (R1)'s Data as Memory Address into AC.

Register mode has indirect mode too.

Register have address data inside, and if you call Instructions, then register calls address data inside the register.

Therefore, effective address is the address in the register.

Example of Register Indirect Mode.

It's load register's data as Memory address, and load it to accumulator.

Relative Addressing Mode

- A method of adding the value of the instruction address field to the content of a specific register to calculate the effective address.
- Certain registers mainly use PC (Program counter)
- Effective address = instruction operand + PC
- Advantage
  - Concise command configuration possible because it can be expressed with a small number of bits

Example

LDA $ADRS   AC ← M[ADRS + PC]

Relative Addressing mode is quiet different.

It has calculation to get address of effective address.
We use Program counter for calculation.

The advantage of relative addressing mode is short to use it.

This is example of relative addressing mode.

Load address of memory, and add with program counter. And store calculation data at accumulator.

Indexed Addressing Mode

- A method of adding the value of the instruction address field to the content of the index register to calculate the effective address.
- Effective address = instruction operand + index register

Example

LDA ADRS(R1) AC ← M[ADRS + R1]

Last, Indexed Addressing mode.
Similar to Relative addressing mode, but it calls Register to calculate effective address.

Effective address can be instrurction operand plus index register.

Example of Indexed Addressing mode.

Get address of memory, and plus register value. And store at accumulator.

Signed / Unsigned Numbers

	Range	Difference	Example
Signed Numbers	$-2^{31} \sim 2^{31}$	First Number can be used as sign.	0000 0000 0000 0000 0000 0000 0000 0001 = 1 1000 0000 0000 0000 0000 0000 0000 0001 = -1
Unsigned Numbers	$0 \sim 2^{32}$	First Number can be used as figure.	0000 0000 0000 0000 0000 0000 0000 0001 = 1 1000 0000 0000 0000 0000 0000 0000 0001 = 4294967297

This is binary numbers for calculate addresses.

What you've heard about signed and unsigned in C, JAVA, or any other high level language.

Use in assembly language, if you input decimal numbers, then it change into binary numbers.

Signed numbers can use in -2 square 31 to 2 square 31, left number can be used as sign. So Allows you to distinguish between negative and positive values.

However, unsigned numbers can use in 0 to 2 square 32. Left number can be used as figure, so allows you to use full size in positive values.

MIPS Assembly Language

- Sample
  - "add a, b, c   # The sum of (b) and (c) is placed in a."
    - Means add (b), (c) and put their sum value in (a).
  - Sharp symbol (#) is comments for human reader.
    - Each line of this language can contain at most one instruction.
    - Comments always terminate at the end of a line.

MIPS Assembly Language is low level language control computer hardwares.

This is sample of MIPS Assembly Language.
"add a, b, c"
It's 3-address format assembly language.

It means add b and c snd put their sum value in (a) address.

Sharp Symbol is comments for human readers.
It's just like \ in C, or # (Hashtag) in python.

Each line of this language can contain at most one instruction.

Comments always terminate at the end of a line.

MIPS Operands

Name	Example	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register $zero always equals 0, and register $at is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

There are two ways in MIPS Operands.

First one is register.
In MIPS, data must be in register to perform arithmetic Instructions like add, sub.
ZERO register always be 0, and $at register is reserved by assembler.

second one is memory words.

MIPS Assembly Language - Arithmetic

Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants

This is arithmetic part of assembly language.
S1, S2, S3 is register operands, and we have to use in MIPS assembly language.

Add immediate is for using at add constants, and the table for add immediate in constants is 20.

MIPS Assembly Language - Data Transfer

Data transfer	load word	`lw $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Word from register to memory
	load half	`lh $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Halfword register to memory
	load byte	`lb $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[`$s2`+20]=`$s1`;`$s1`=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	`$s1` = $20 * 2^{16}$	Loads constant in upper 16 bits

This is data transfer instructions.

It can use memory address or registers.

Using unsigned methods can add ‘u’ in instructions, and you can use word, half, byte.

MIPS Assembly Language - Logical

Logical	and	`and $s1,$s2,$s3`	`$s1` = `$s2` & `$s3`	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	`$s1` = `$s2` | `$s3`	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	`$s1` = ~ (`$s2` | `$s3`)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	`$s1` = `$s2` & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	`$s1` = `$s2` | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	`$s1` = `$s2` << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	`$s1` = `$s2` >> 10	Shift right by constant

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bit-by-bit AND	&	&	`and`, `andi`
Bit-by-bit OR	|	|	`or`, `ori`
Bit-by-bit NOT	~	~	`nor`

Logical Operations can call as shift.
They use in binary shifts.

Assembly use binary numbers to calculate, so it’s very useful.

MIPS Assembly Language - Conditional Branch

Conditional branch	branch on equal	`beq $s1,$s2,25`	if (\$s1 == \$s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (\$s1!= \$s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant unsigned

Conditional Branch can use in making decisions.

You can use branch as if, else, loop in like C language.

MIPS Assembly Language - Unconditional Jump

Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

This is jump method.

In c, you can use goto() functions to move lines, it’s same function as C.

It’s useful in functions, like sort function or using at loop section.

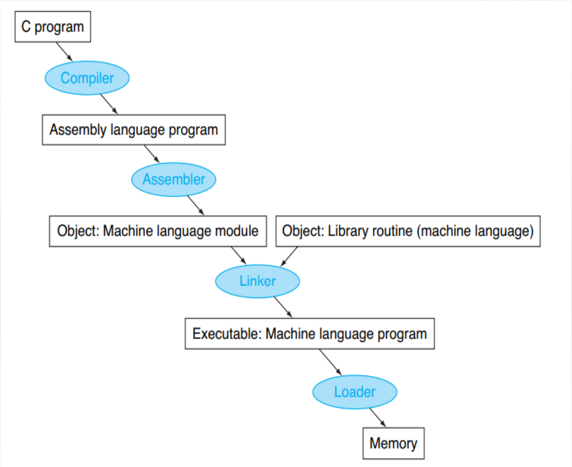
Synchronization

- Parallel execution is easier when tasks are independent, but often they need to cooperate.
- If they don't synchronize, there is a danger of a data race, where the results of the program can change depending on how events happen to occur.
- In computing, synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions.

Translating and Starting Program

Elements

- Compiler
- Linker
- Loader
- Dynamically Linked Libraries(DLLs)



Compiler

- A Compiler is primarily used for programs that translate source code from a high-level programming language to a machine level language to create an executable program.
- Previously, many operating systems and assemblers were written in assembly language because memories were small, and compilers were inefficient.
- Optimizing compilers today can produce assembly language programs nearly as good as an assembly language expert, and sometimes even better for large programs.

Assembler

- An assembler is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations. Some people call these instructions assembler language and others use the term assembly language.
- Assembler in assembly language simplifies the translation and programming.
- Common variations of machine language instructions, which assemblers can use as their own is called pseudoinstructions.
- pseudoinstructions give MIPS a richer set of assembly language instructions than those implemented by the hardware.

Assembler

- Assemblers keep track of labels used in branches and data transfer instructions in a symbol table. As you might expect, the table contains pairs of symbols and addresses.
- Symbol table A table that matches names of labels to the addresses of the memory words that instructions occupy.
- The table contains pairs of symbols and addresses

Assembler

The object file for UNIX systems typically contains six distinct pieces:

- The object file header describes the size and position of the other pieces of the object file.
- The text segment contains the machine language code.
- The static data segment contains data allocated for the life of the program.
- The relocation information identifies instructions and data words that depend on absolute addresses when the program is loaded into memory.
- The symbol table contains the remaining labels that are not defined, such as external references.
- The debugging information contains a concise description of how the modules were compiled so that a debugger can associate machine instructions with C source files and make data structures readable.

Linker

A systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file.

- There are three steps for the linker:
    I. Place code and data modules symbolically in memory.
    II. Determine the addresses of data and instruction labels.
    III. Patch both the internal and external references.
- The linker produce an executable file which has a similar format type as an object file

Loader

A systems program that places an object program in main memory so that it is ready to execute.

The loader follows these steps in UNIX systems:

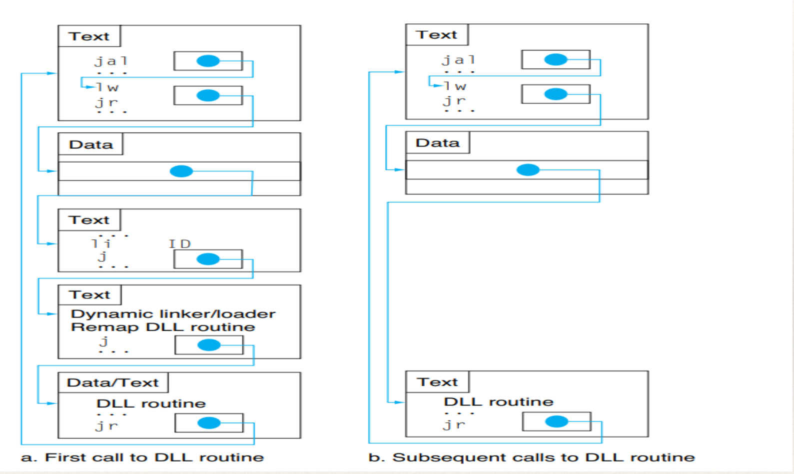
- Reads the executable file header to determine size of the text and data segments.
- Creates an address space large enough for the text and data.
- Copies the instructions and data from the executable file into memory.
- Copies the parameters (if any) to the main program onto the stack.
- Initializes the machine registers and sets the stack pointer to the first free location.
- Jumps to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program. When the main routine returns, the start-up routine terminates the program with an exit system call.

DLL

Dynamically Linked Libraries (DLLs) Library routines that are linked to a program during execution.

- In the initial version of DLLs, the loader ran a dynamic linker, using the extra information in the file to find the appropriate libraries and to update all external references.
- The lazy procedure linkage version of DLLs, where each routine is linked only after it is called.

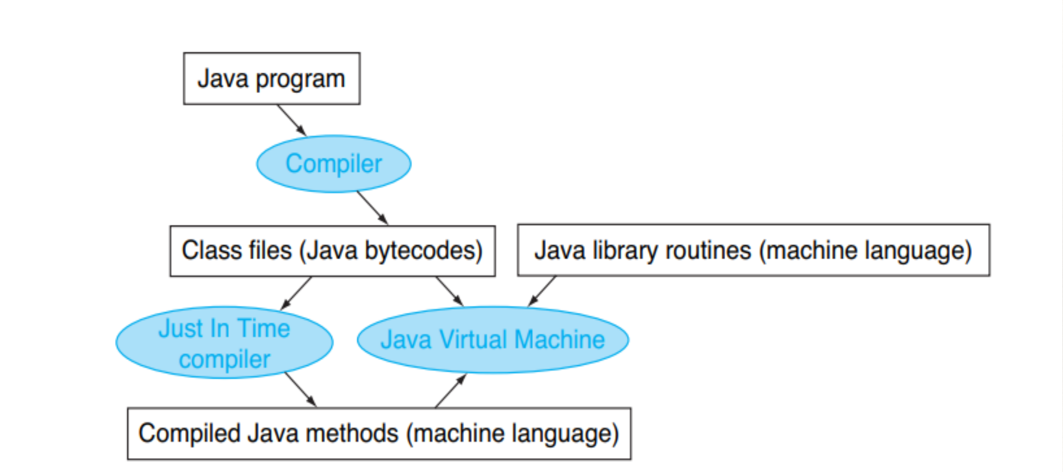
DLL


Text
jal
...
lw
jr
...
Data
Text
...
li ID
j
...
Text
Dynamic linker/loader
Remap DLL routine
j
...
Data/Text
DLL routine
...
jr
a. First call to DLL routine
Text
jal
...
lw
jr
...
Data
Text
DLL routine
...
jr
b. Subsequent calls to DLL routine

Starting a JAVA Program

- Java was invented with a different set of goals. One was to run safely on any computer, even if it might slow execution time.
- Rather than compile to the assembly language of a target computer, Java is compiled first to instructions that are easy to interpret: the Java bytecode instruction set.
- Like the C compiler, the Java compiler checks the types of data and produces the proper operation for each type.

Start a JAVA Program



Advantages

- Ease of writing an interpreter
- Better error messages
- Smaller object code
- Machine independence

Array VS Pointer

- Example
  - ```
    clear1(int array[], int size)
    {
      int i;
      for (i = 0; i < size; i += 1)
          array[i] = 0;
    }
    clear2(int *array, int size)
    {
      int *p;
      for (p = &array[0]; p <
     &array[size]; p = p + 1)
           *p = 0;
    }
    ```
  - `Clear1` uses array, while `clear2` uses pointers.
    ```

JAVA and C

- People used to be taught to use pointers in C to get greater efficiency than that available with arrays: “Use pointers, even if you can’t understand the code.”
- Modern optimizing compilers can produce code for the array version that is just as good.
- Most programmers today prefer that the compiler do the heavy lifting.

ARM processor

	ARM	MIPS
Date announced	1985	1985
Instruction size (bits)	32	32
Address space (size, model)	32 bits, flat	32 bits, flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Integer registers (number, model, size)	15 GPR × 32 bits	31 GPR × 32 bits
I/O	Memory mapped	Memory mapped

ARM is the most popular instruction set architecture for embedded devices, like smartphones.

ARM is Advanced RISC Machine, ARM came out the same year as MIPS and followed similar philosophies.

The principle difference is that MIPS has more registers and ARM has more addressing modes.

There is a similar core of instruction sets for arithmetic-logical and data transfer instructions for MIPS and ARM.

Instructions in ARM processor

	Instruction name	ARM	MIPS
Register-register	Add	add	addu, addiu
	Add (trap if overflow)	adds; swivs	add
	Subtract	sub	subu
	Subtract (trap if overflow)	subs; swivs	sub
	Multiply	mul	mult, multu
	Divide	—	div, divu
	And	and	and
	Or	orr	or
	Xor	eor	xor
	Load high part register	—	lui
	Shift left logical	lsl[1]	sllv, sll
	Shift right logical	lsr[1]	srlv, srl
	Shift right arithmetic	asr[1]	srav, sra
	Compare	cmp, cmn, tst, teq	slt/i, slt/iu
Data transfer	Load byte signed	ldrsb	lb
	Load byte unsigned	ldrb	lbu
	Load halfword signed	ldrsh	lh
	Load halfword unsigned	ldrh	lhu
	Load word	ldr	lw
	Store byte	strb	sb
	Store halfword	strh	sh
	Store word	str	sw
	Read, write special registers	mrs, msr	move
	Atomic Exchange	swp, swpb	ll;sc

Addressing mode	ARM v.4	MIPS
Register operand	X	X
Immediate operand	X	X
Register + offset (displacement or based)	X	X
Register + register (indexed)	X	—
Register + scaled register (scaled)	X	—
Register + offset and update register	X	—
Register + register and update register	X	—
Autoincrement, autodecrement	X	—
PC-relative data	X	—

These are ARM Processors instructions.

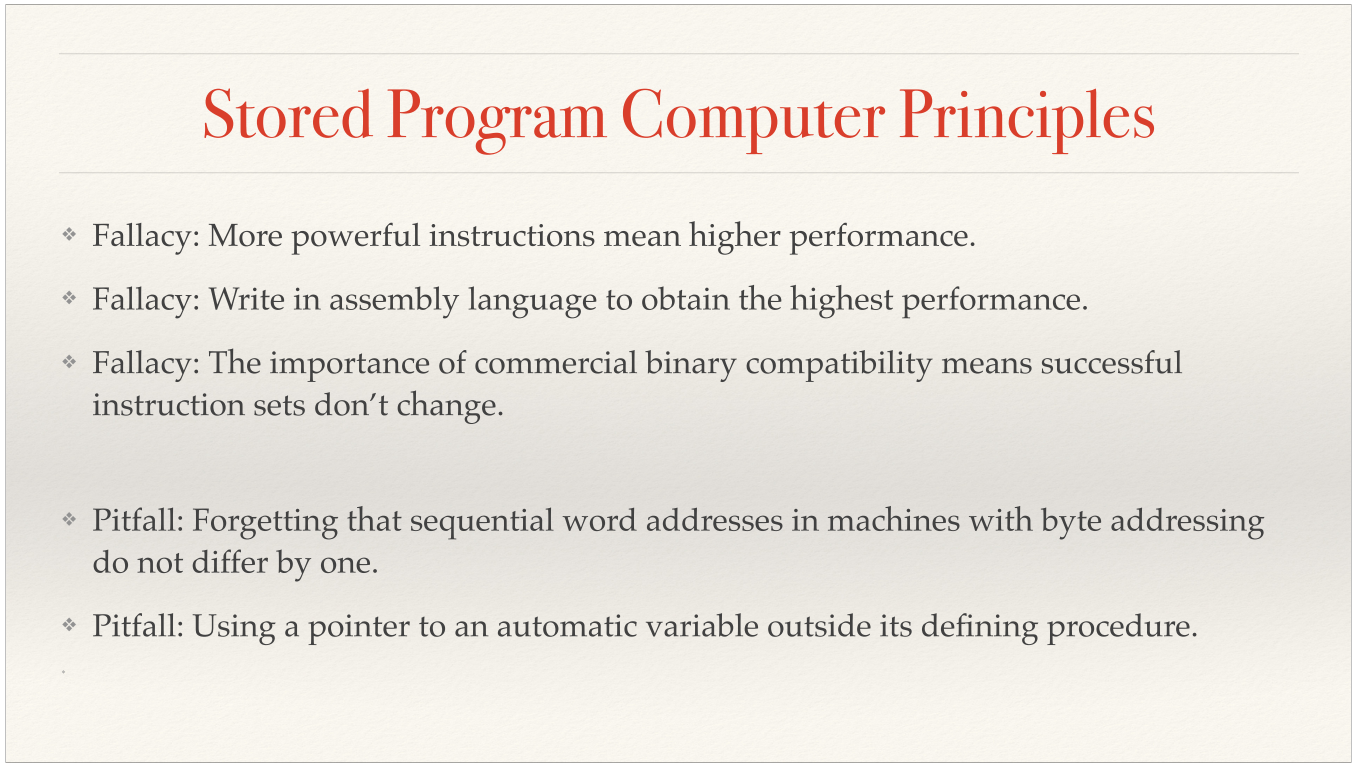
Quite similar with MIPS processor instructions.

Unlike MIPS, ARM doesn’t have $zero reserved registers.

And Memory addressing modes in ARM has 9, and MIPS addressing modes are 3.

These are addressing mode that supports arm and mips.

ARM has separate register indirect and register + offset addressing modes, rather than just putting 0 in the offset of the latter mode. To get greater addressing range, ARM shifts the offset left 1 or 2 bits if the data size is halfword or word.



Stored Program Computer designing has fallacy and pitfalls.

Fallacies for designing stored program computers, first one is ‘More powerful instructions mean higher performance.’
However, powerful instructions tend to be complex and difficult to execute in hardware.
This penalizes all commands.

Second one is ‘write in assembly language to obtain the highest performance’
In the old days, it was true.
Because back then, there was no guarantee that compilers would produce fast code.
But modern compilers know what code to generate for modern processors.

Third one is ‘The importance of commercial binary compatibility means successful instruction sets don’t change.’
However, new commands are slowly emerging. That said, the x86, ARM instruction set is getting bigger and bigger.
It's nice to have backwards compatibility, but you have to put up with a huge instruction set.

Pitfalls have two.
Forgetting that sequential word addresses in machines with byte addressing do not differ by one.
It’s an error that find the address of the word by passing the address of the register one more space.

Using a pointer to an automatic variable outside its defining procedure.

Pointers to automatic variables can lead to chaos.

Design Principles

- Simplicity favors regularity.
- Smaller is faster.
- Make the common case fast
- Good design demands good compromises.

Away from pitfalls and fallacy, we should think about design principles.
These are four design principles for architecture.

Simplicity favors regularity, smaller size of design will be faster, make the common cases fast, good design demands good compromises.

Source

- COMPUTER ORGANIZATION AND DESIGN (4th Edition)
  David A. Patterson, John L. Hennessy