Outline of Lecture

1. Representing Instructions in the Computer

2. Instructions for Making Decisions

Representing Instructions in the Computer

- Numbers are kept in the computer hardware as a series of high and low electronic signals - just two choices.
- Naturally, they are considered ***base 2*** numbers. They are called ***binary*** numbers.
- Like humans - we are familiar with ***base 10*** numbers (10 fingers). These numbers are called ***decimal*** numbers.
- As a result, a binary number or ***digit*** is thus the fundamental unit of computing.
- All information (data and programs) is composed of binary digits or ***bits***.

- Instructions (e.g., `lw`, `add`) are stored in the computer as a series of high and low electronic signals - thus they can be represented as binary numbers.

- Each piece of information of an instruction is represented by a binary number.

- The MIPS instruction `add $t0, $s1, $s2` is represented as follows:

0	17	18	8	0	32

- **Each of these segments of an instruction is called a *field*.**

  ➜ **The first and last fields (containing 0 and 32) in *combination* tell the MIPS computer to perform addition.**

  ➜ **The second field (containing 17) gives the number of the register that is the *first source operand* of the addition operation ($17).**

  ➜ **The third field (containing 18) gives the number of the register that is the *second source operand* of the addition operation ($18).**

  ➜ **The fourth field (containing 8) gives the number of the register that is to receive the sum ($8) - *destination* register.**

  ➜ **The fifth field (containing 0) is not used in this instruction - it will be used in other instructions.**

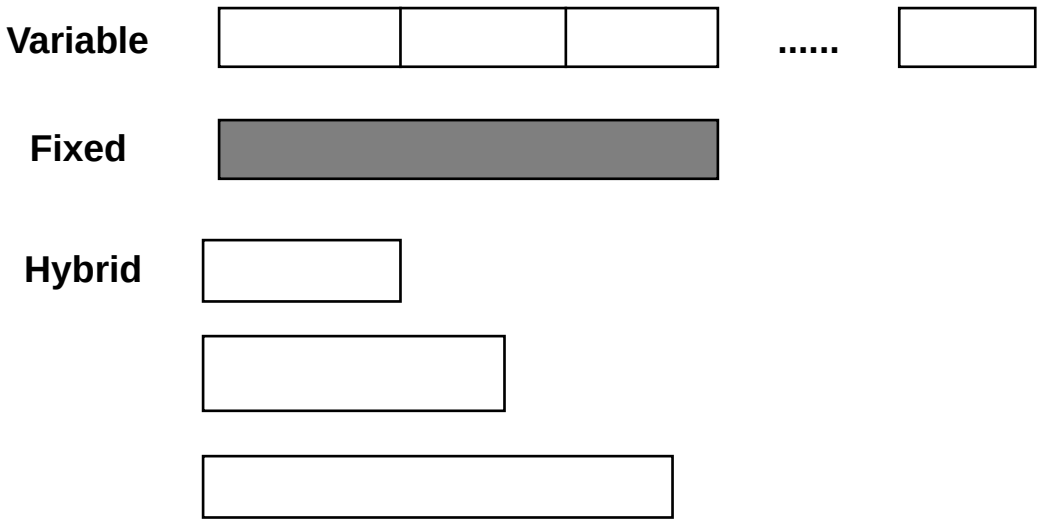
- Inside the computer the instruction `add $t0, $s1, $s2`, is represented using binary numbers as follows:

0	17	18	8	0	32
000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- This layout of the instruction is called ***instruction format***.

- Each MIPS instruction is 32 bits - the same size as a data word.

There are three ways of encoding an instruction in a computer (e.g., having an instruction format)



∴ **If code size is most important, use variable length instructions.**

∴ **If performance is most important, use fixed length instructions.**

MIPS design always favors simplicity - thus they use fixed size formats for instructions.

To be simple, you have to be regular (no special cases or very few of them) - The format of instructions in MIPS is regular.

This will have many effects on the processor design (how complex - fast - powerful, etc.) as will be seen during this course.

R-Type Instructions

- The general format layout of MIPS computer instructions is given as follows - called ***R-type***:

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

- The meaning of each field of MIPS instructions is:

  ➜ ***op:*** operation for the instruction.

  ➜ ***rs:*** the first register source operand.

  ➜ ***rt:*** the second register source operand.

  ➜ ***rd:*** the register destination operand; it gets the results of the operation.

  ➜ ***shamt:*** shift amount (will be explained later).

  ➜ ***funct:*** function; this field selects the variant of the operation in the *op* field.

- If we have a `lw` or `sw` instruction

`lw $t0, 32($3)`,

then we need to specify the ***address*** (in this case 32) within the instruction format.

- If we use one of the source operand fields from the previous instruction format (*rs* or *rt*), then the instruction would be limited to just $2^5 = 32$ memory locations - which is ***too small***.

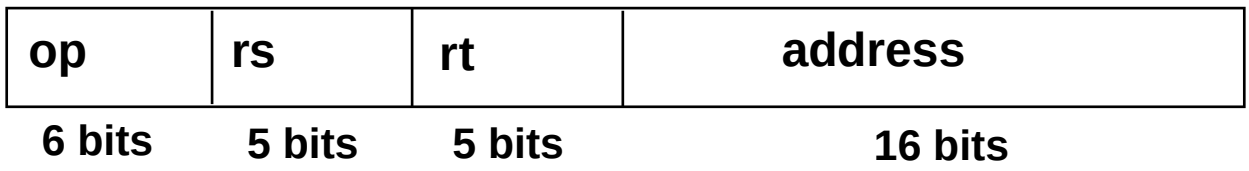
- There is a trade-off between keeping the size of the instructions fixed and having a single instruction format.

Principle #3: Good design requires compromises.

MIPS keeps the length of instructions the same, but instructions could have different formats.

I-Type Instructions

- The second type of instruction format for a MIPS computer is for data transfer instructions (e.g., `lw` and `sw`), and is as follows - called *I-type*:



Example:

```
lw $t0, 32($s3)
```

`19` is placed in `rs` field

`8` is placed in the `rt` field (unlike the previous format, `rt` is the destination register in this case)

`32` is placed in the `address` field.

- **The two different types of instructions are distinguished from each other by the *op* field.**

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

Example

Let us take an example all the way from what the programmer writes to what the machine executes. Assuming that $t1 has the base address of the array A and $s2 corresponds to h, the C assignment statement

```
A[300] = h + A[300];
```

is compiled into:

```
lw $t0, 1200($t1)    # Temporary reg $t0 gets A[300]
add $t0, $s2, $t0    # Temporary reg $t0 gets h+A[300]
sw $t0, 1200($t1)    # Stores h+A[300] back into A[i]
```

What is the MIPS machine language code for these 3 instructions?

Answer

Assume that the starting location or *address* for array A is 1200 in base 10 or (0000 0100 1011 0000 base 2).

Then the machine language for the 3 instructions are:

op	rs	rt	(rd)	(shamt)	address /funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

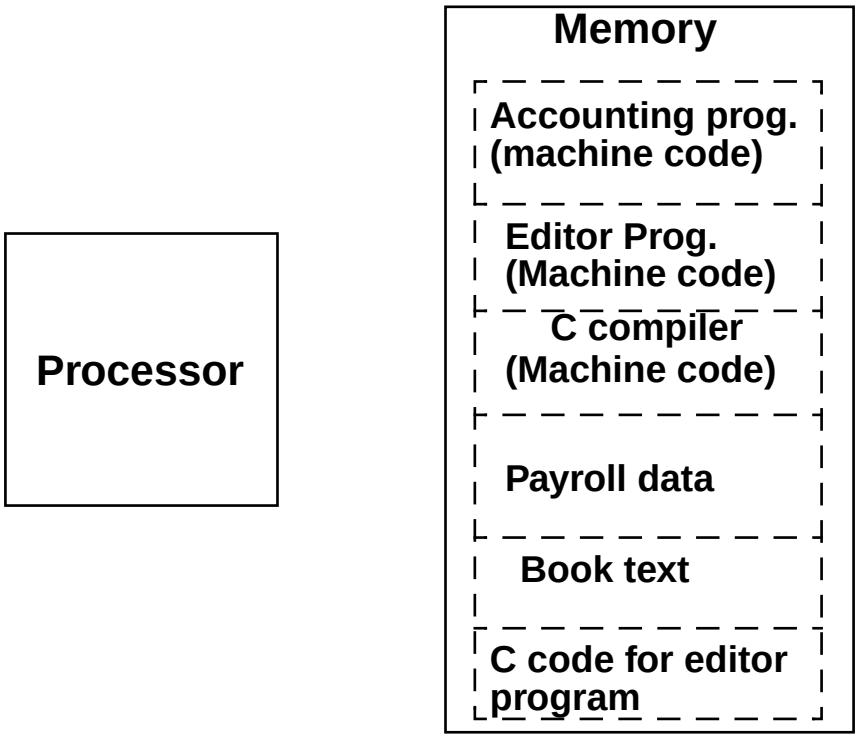
The binary equivalent of the above table is:

op	rs	rt	(rd)	(shamt)	address /funct
100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

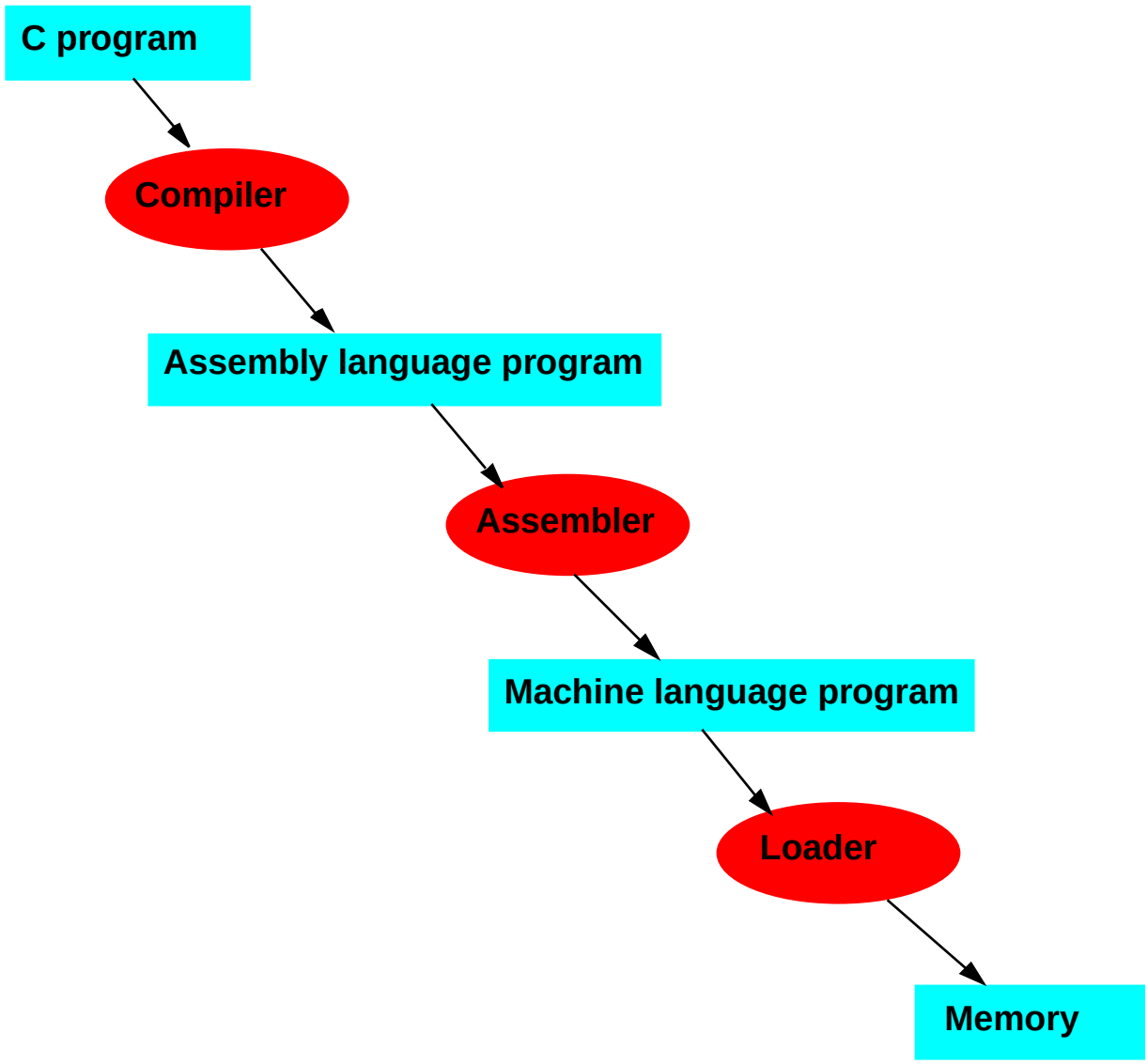
The Big Picture

Today's computers are built on two key principles:

1. **Instructions are represented as numbers; and**
2. **Programs can be stored in memory to be read or written just like numbers.**



The various *layers* a high-level language program has to go through to get executed.



Further Reading

Chapter 3 and Appendix A. **David A. Patterson and John L. Hennessy. *Computer Organization & Design: The Hardware / Software Interface*. Morgan Kaufman Publishers, 1998.**