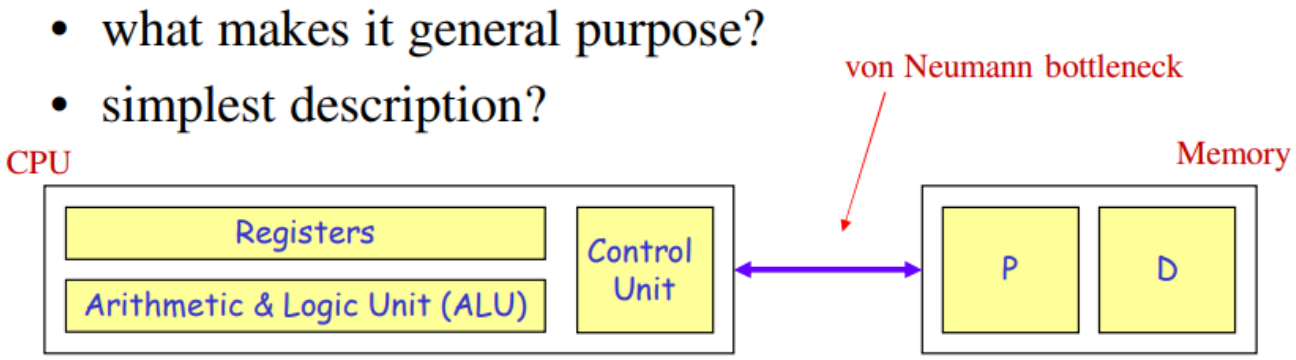
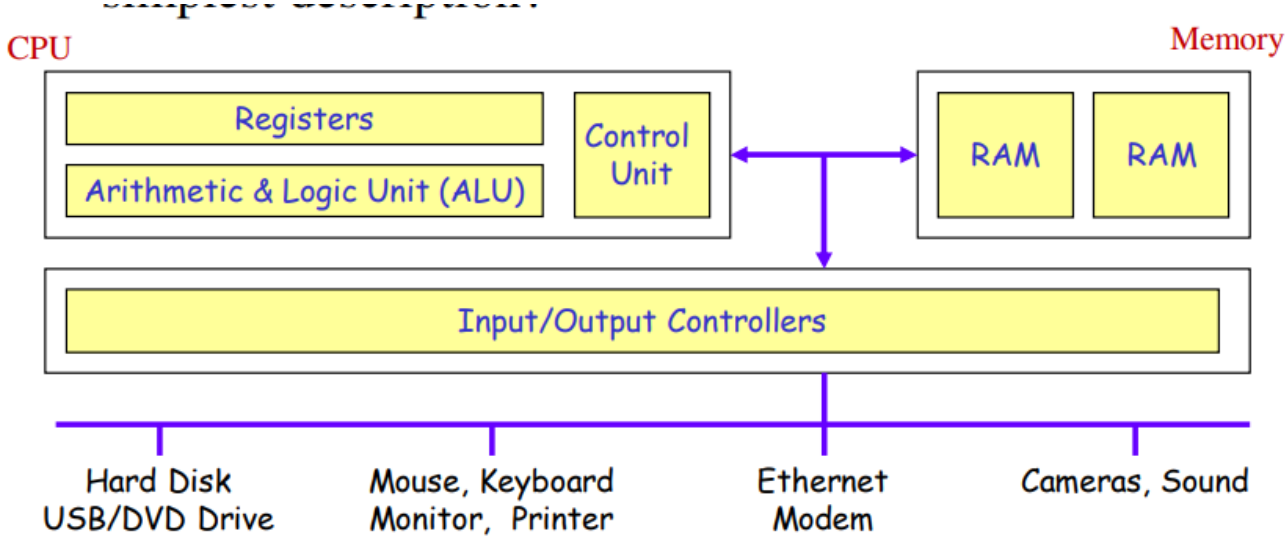
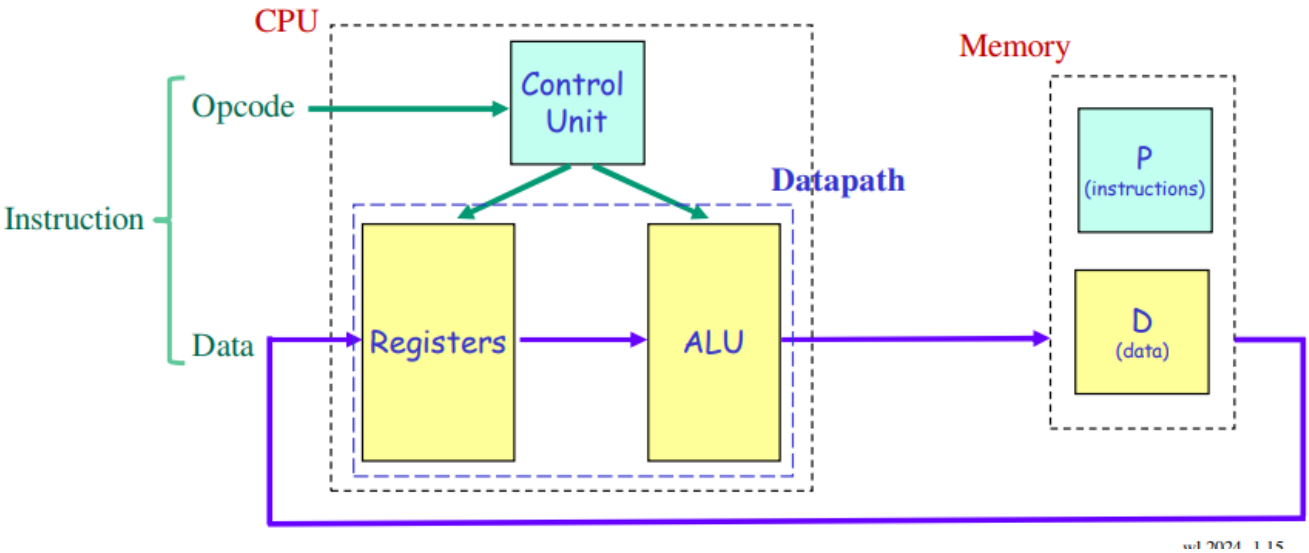
Lecture 1, Instruction Format



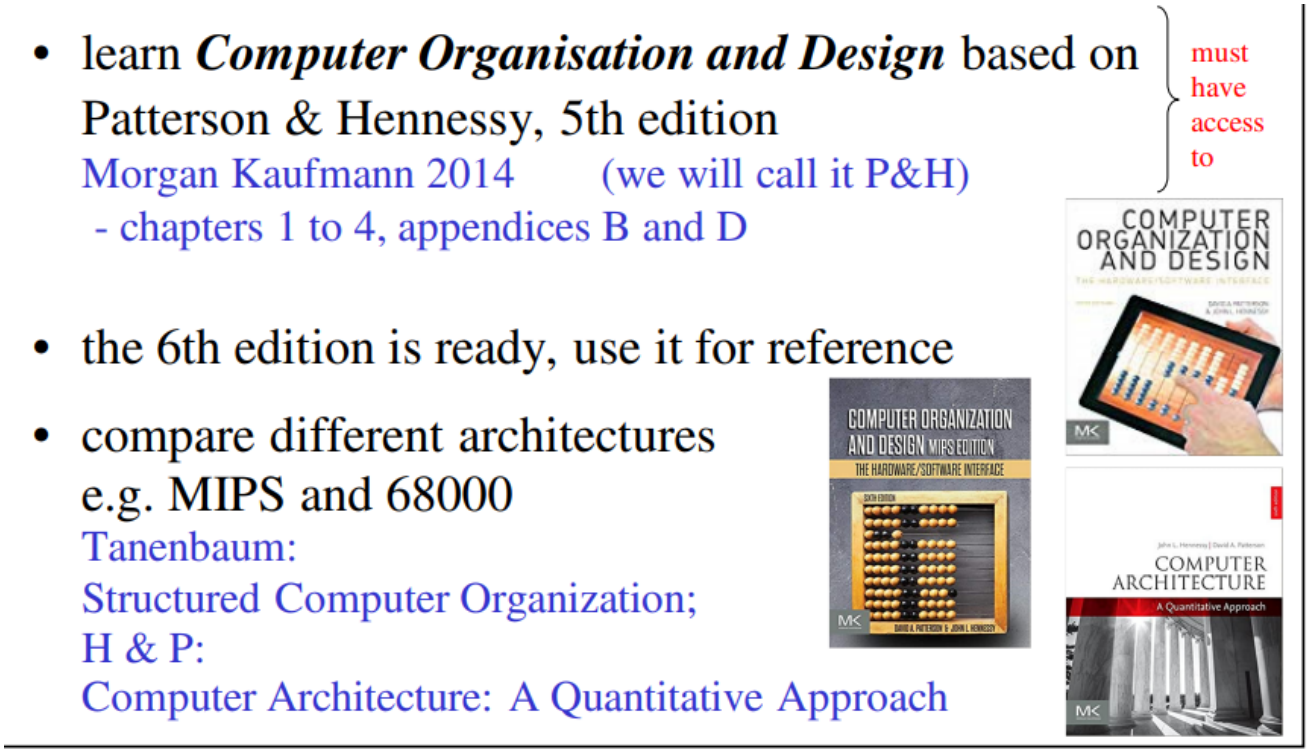
We can begin to fill in the details. The block on the left is often known as the Central Processing Unit, CPU. The CPU contains registers as fast memory for storing data, an ALU for computation, and a Control Unit for managing the operations. The block on the right is the memory, containing programs P and data D. Storing programs in memory allows the CPU to become general purpose: by supplying different programs to the CPU, it can perform different computations. The connection between the CPU and the memory is called the von Neumann bottleneck by John Backus, who invented Fortran but later was attracted to functional programming.



Lets add the remaining resources: those for input/output. Most computers follow this architecture, which is sometimes called the von Neumann architecture, after one of the computing pioneers, John von Neumann. But is it the most efficient?

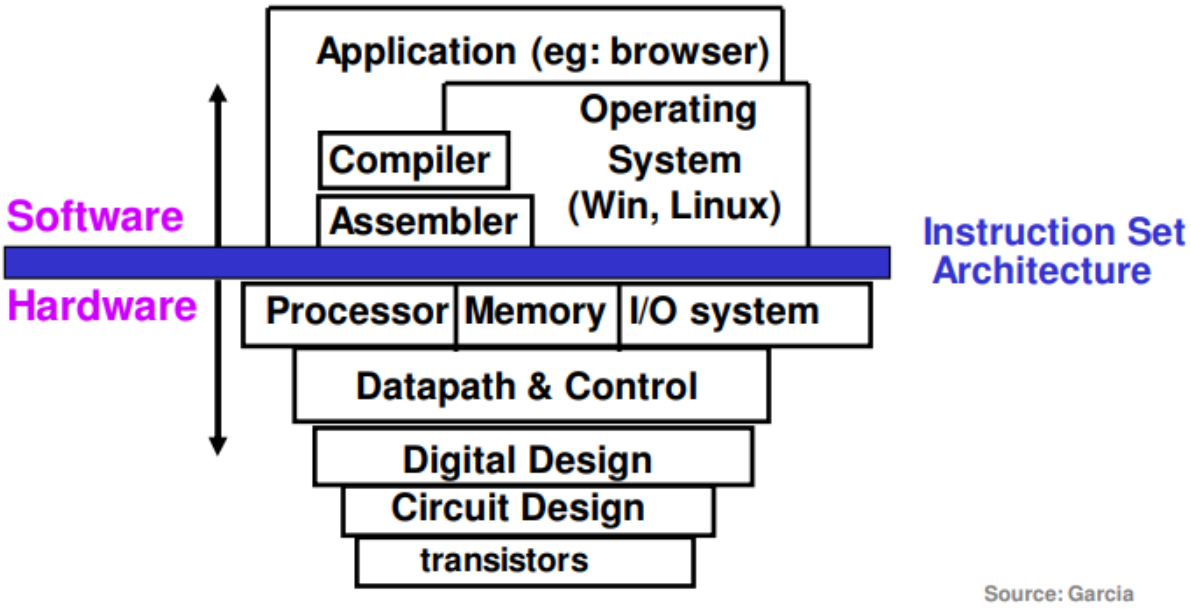


Before we describe the topics covered by this module, let us look at what is a datapath. The datapath of a processor is just the path for processing data from the memory. If we re-arrange our simple diagram of a computer, then the datapath consists mainly of the registers and the ALU, responsible for processing data fetched from the data memory D and storing results back to D, supervised by the control unit. The control unit follows the Opcode of the instructions fetched from the program memory P.



What is Computer Architecture?

Architecture = Instruction set architecture (ISA) + machine organisation
ISA examples: x86, ARM, MIPS, SPARC, RISC-V



The ISA is an abstraction of the hardware resources of a given machine. It forms the lowest level of software, on which higher levels of software can be built. Note that one ISA can be implemented in many ways. For example, both Intel and AMD supports the x86 ISA.

Complex Instruction Set Computers, CISC

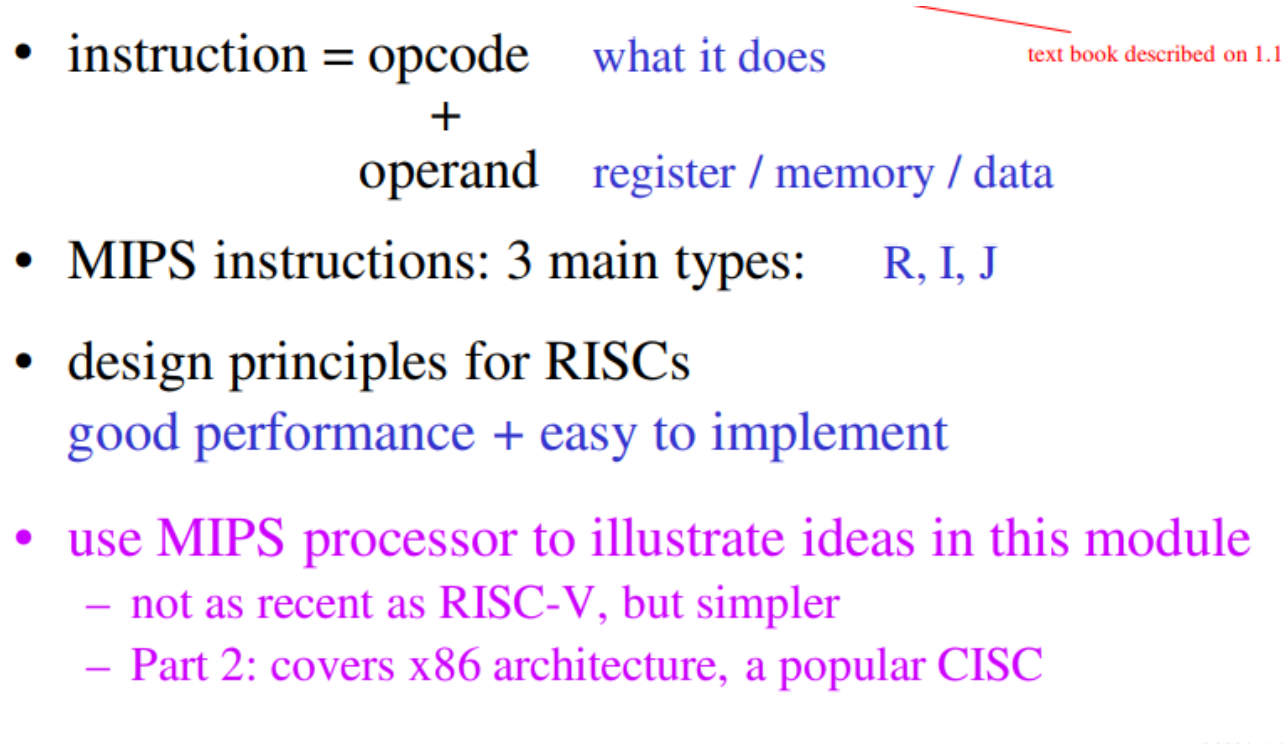
- dense code
- simple compiler
- powerful instruction set
- variable format

Reduced Instruction Set Computers, RISC

- simple instructions
- fixed format
- optimising compiler
- speed
- low development cost
- adapt to new technology

There are two main ISA approaches: CISC and RISC. CISC includes more complex instructions, which reduces the workload of compilers (which translates

high-level languages like C into the assembly language). However, this also means that CISC instructions take more time to execute, as they are more complex. Since the compilers are becoming better, RISC is becoming faster and more adaptive to new technologies. You will learn the details of their differences in the third lecture. Most ISAs are now RISC based, except the x86 ISA. Part one of the module will cover RISC, while Part 2 will cover CISC.



instruction = opcode (what it does) + operand (register / memory / data)

MIPS instructions:

- 3 main types: R, I, J
- design principles for RISCs
  - good performance
  - easy to implement
- use MIPS processor to illustrate ideas in this module
  - not as recent as RISC-V, but simpler
  - Part 2: covers x86 architecture, a popular CISC

An instruction for a processor usually has two parts: opcode and operand. Opcode specifies its function, while the operand specifies the information or data needed to carry out that function. For example, a well-known processor called MIPS has 3 main instruction types, which will be detailed later.

MIPS architecture

- representative of modern RISC architectures
- 32 registers $0..$31 32 bits each
- $0 wired to 0, the others general-purpose
- register-register or load-store architecture
  - most instructions involve registers only: fast
    ```
    add $1, $2, $3        # reg1 = reg2 + reg3
    ```
    comment
  - special memory access instructions: possibly multicycle
    ```
    lw $8, Astart($19)    # reg8 = M[Astart + reg19]
    ```
- goal: minimise memory access; why?

MIPS has a simple architecture, with 32 registers, each of 32 bits. Most instructions related to computation involve only registers and not memory, since registers are much faster than memory. There are special instructions for memory access which are relatively slow; so once data are brought into the CPU, they would stay there as long as possible before being sent back to memory.

MIPS instructions: R-type

- 3 types: R-type (register), I-type (immediate), J-type (jump) } fixed size: 32 bits
- R-type: arithmetic, comparison, logical, …

```
add $8, $17, $18        # reg8 = reg17 + reg18
```

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
0	17	18	8	0	32
opcode	source 1	source 2	dest.	shift	function

- MIPS format: usually destination comes first

The three types of MIPS instructions are R, I and J. The R-type covers instructions for computation, including arithmetic and logical operations.

MIPS instructions: I-type

- immediate (I-type): memory access, conditional branches, arithmetic involving constants
- memory access:

```
lw $8, Astart($19)   # reg8 = M[Astart + reg19]
```

6 bits	5 bits	5 bits	16 bits
35	19	8	Astart
opcode	source	dest.	immediate constant

- arithmetic:

```
addi $1, $2, 100   # reg1 = reg2 + 100
```

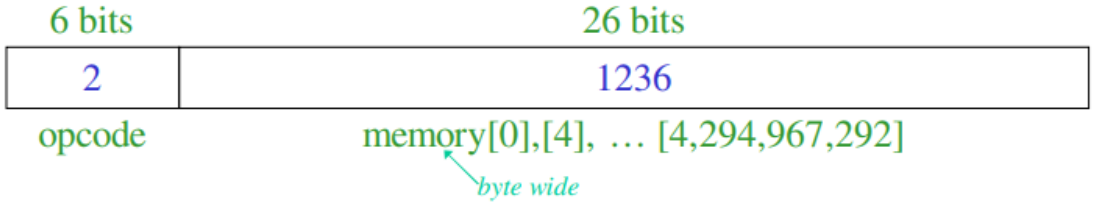
- Immediate (I-type):
  - memory access
  - conditional branches
  - arithmetic involving constants
- memory access: `lw $8, Astart($19)` (`# reg8 = M[Astart + reg19]`)
- arithmetic:
  - `addi $1, $2, 100` (`# reg1 = reg2 + 100`)

The I-type instructions cover memory access and conditional branching. It can be used for arithmetic when, for example, a constant value is involved.

MIPS instructions: J-type

- jump (J-type): unconditional jump to instruction in memory

```
j 1236          # jump to instruction at address 1236
```



- jal: jump and link `# save address of next instruction # in register before jumping`
- "jump" instructions can be I-type or R-type

```
I-type : bne $19,$20,Label # if reg19 ≠ reg20 goto Label
R-type: jr  $ra            # jump to address in register ra
```

wl 2024 1 30

- jump (J-type): unconditional jump to instruction in memory
- `jal` : jump and link (# save address of next instruction, # in register before jumping)
- "jump" instructions can be I-type or R-type
  - I-type : `bne $19,$20,Label # if reg19 ≠ reg20 goto Label`
  - R-type: `jr $ra # jump to address in register ra`

J-type instructions covers unconditional jumps to instructions in memory. The `jal` instruction enables returning to the instruction right after the `jal` instruction.

However, note that “jump” instructions can be I-type or R-type.

Example

- if (i = j) f = g+h; else f = g-h;
- allocate reg16 = f reg17 = g reg18 = h
  reg19 = i reg20 = j
-
```
        bne $19, $20, Else     # if i ≠ j goto Else
        add $16, $17, $18      # f = g+h      (if i = j)
        j   Exit               # goto Exit
Else:   sub $16, $17, $18      # f = g-h      (if i ≠ j)
Exit:
```
- while-loop: similar

This example shows how to assign data to registers to initialise them, so that they can then be used to implement computations.

Remarks

- only 2 conditional branches, bne and beq
- need slt (set on less than)
```
slt $1, $16, $17    # if reg16 < reg17 then reg1 = 1
                    #                  else reg1 = 0
```
- implement branch to L on reg16 < reg17 as
```
slt $1, $16, $17    # … if reg1 ≠ 0 then goto L
bne $1, $0, L       #     (reg0 always 0)
```
- load constant hex 000A000B to register 5, use load upper/lower immediate (lui/lli)
```
lui  $5, 10         # reg5 = 000A0000
addi $5, $5, 11     # reg5 = reg5 + 000B
```

wl 2024_1.32

MIPS has relatively few types of instructions, so some tasks would need two instructions to complete rather than one; e.g. it only has two instructions covering conditional branches. To implement some conditionals, two instructions are needed to make use of the SLT instruction to compute “less than”. As another example, initialising a register would need 2 instructions, `lui` and `addi`. We shall continue our discussions about the MIPS processor in the next few lectures.