Architecture et Système

Stefan Schwoon

Cours L3, 2014/15, ENS Cachan

Processor architecture

Hardwired

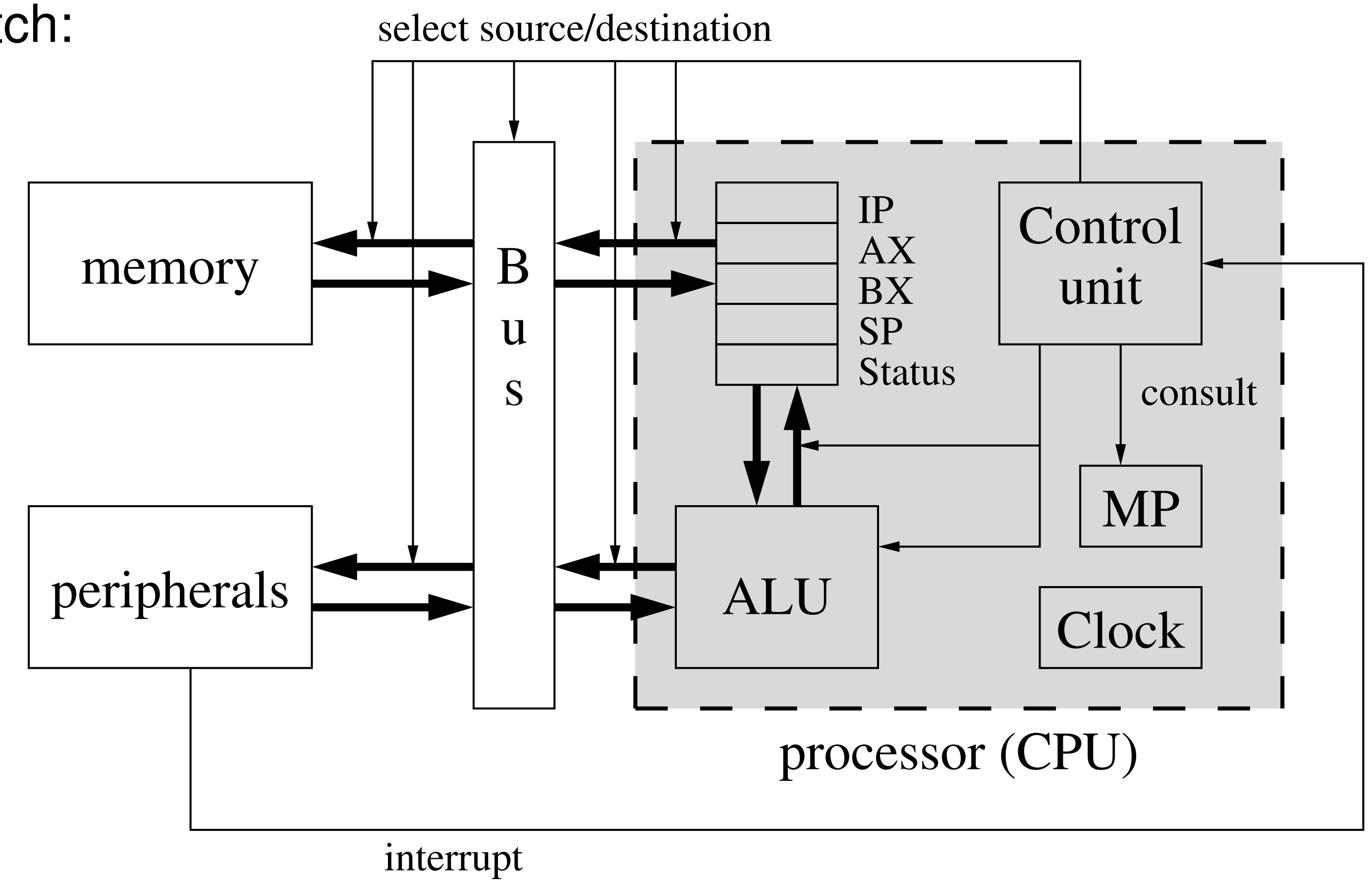
Microprogrammed

CISC / RISC (Complex / Reduced instruction set computer)

Explicit Parallelism (EPIC / VLIW)

Recall: Basic architecture

A first sketch:



Data path and control unit

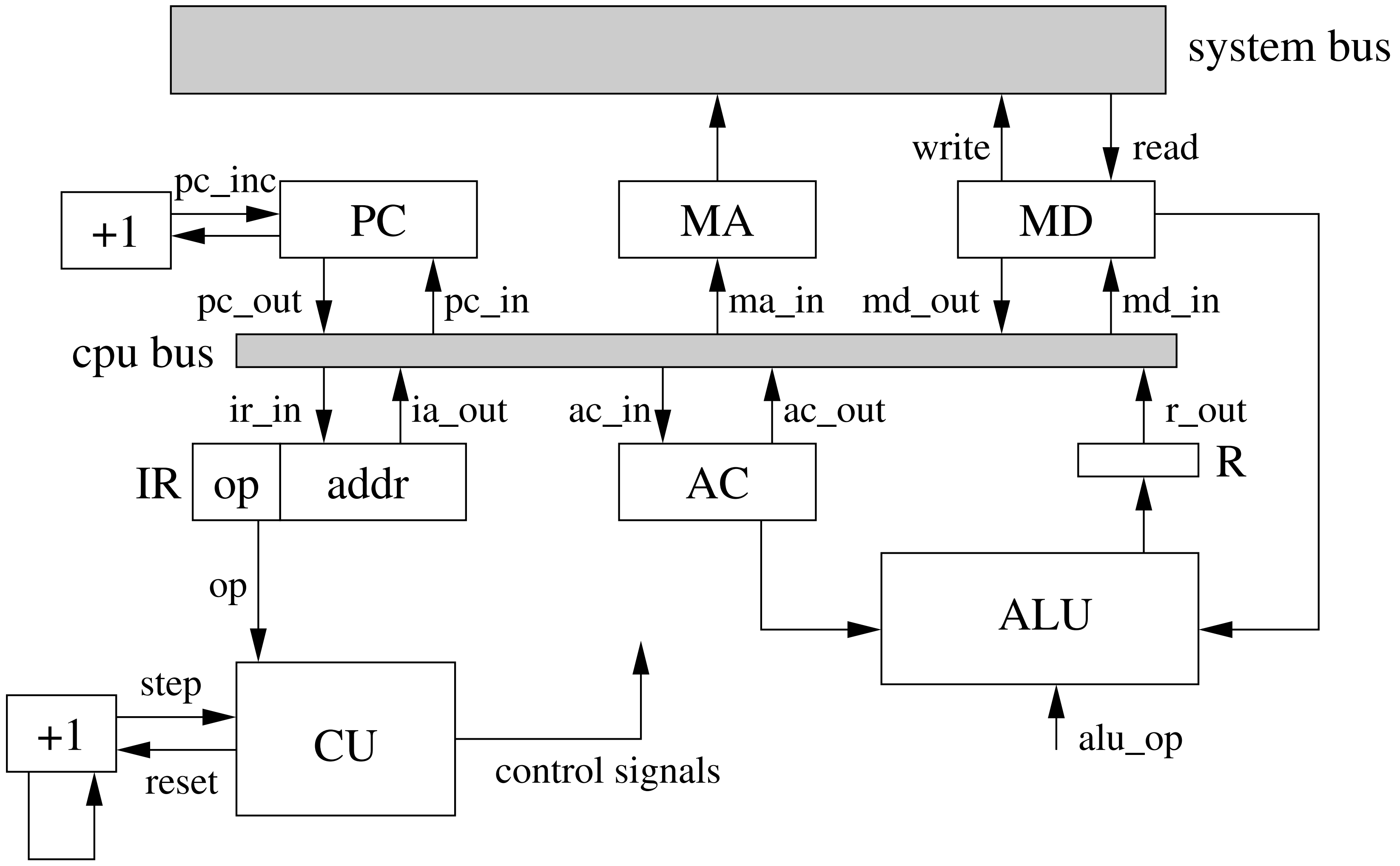
Processor can be said to consist of two parts:

Data path: everything that stores/manipulates data, e.g. registers, ALU, bus, . . .

Control: responsible for organising transfers inside the data path; control unit, control signals, multiplexer/decoders, . . .

Most components have both aspects, e.g., a bus contains data lines co-ordinated by control signals.

Sample CPU design



This design has a CPU-internal bus and a system bus.

MA/MD communicate with the memory via system bus.

IR: instruction register / MA: memory address / MD: memory data

Input/output to buses controlled by the named signals written next to them;
these are provided by the CU.

How the CU works

Input (in this example): op (operation code) / step (clock phase)

To execute the next operation, we fetch it from memory (IF):

> phase 0: load PC into MA, increase PC (`pc_out`, `ma_in`, `pc_inc`)
>
> phase 1: fetch operation from memory (`read`)
>
> phase 2: move operation to IR (`md_out`, `ir_in`)

Phase 3 and following serve to execute the operation.

Note: This could be sped up, e.g., if we allow PC to access the system bus directly.

Executing an operation

Let us regard two operations:

LOAD *addr*: load content of addr into accumulator

- phase 3: load *addr* into MA: (`ia_out`, `ma_in`)
- phase 4: fetch data from memory (`read`)
- phase 5: transfer to ALU, next op (`md_out`, `alu_in`, `reset`)

ADD *addr*: add content of addr to accumulator

- phase 3: load *addr* into MA: (`ia_out`, `ma_in`)
- phase 4: fetch data from memory (`read`)
- phase 5: perform addition (`alu_op=add`)
- phase 6: transfer to ALU, next op (`r_out`, `alu_in`, `reset`)

Control signals

How to obtain the correct control signals:

Hardwired:

Build a logical circuit for each signal, with inputs op and phase.

Used in first computer designs.

Microprogramming:

Use op and phase as index into a ROM, which delivers the values of all signals.

(needs additional mechanisms for conditional branching)

Very popular until 1970s/80s.

Advantages: ROM faster than memory / complex instructions possible (ease for compilers) / flexible (can exchange ROM if needed)

Example: Intel instruction set

Complex instruction set, with powerful instructions (including loops)

Operands with 8/16/32 bits

Can use part or all of registers

Varying length of instructions (1-7 bytes), requires additional instruction decode phase

Requires more complex control

CISC = complex instruction set computer

RISC architecture

Starting from 1970s, becomes popular in the 1990s (faster memory, more transistors possible)

RISC = reduced instruction set computer

idea: only provide basic, most necessary operations in CPU
(shift complexity from microcode to assembly code)

better speed thanks to:

- every instruction executable in one clock cycle, thanks to hardwired logic
- allows to interleave phases of executions (IF/ID/EX), multiple instructions executed at once (superscalar)
- wider, regular instruction words: opcode always of same length, operands always at same position $\Rightarrow$ simpler, faster control logic
- more registers available

Issues in superscalar architecture

CPU tries to execute the next *n* instructions in parallel

Problems due to data dependencies and branching

Solutions: speculative execution, branch prediction

EPIC / VLIW

EPIC = explicitly parallel instruction computing

VLIW = very long instruction word

Instruction word contains several instructions at one, which can be executed in parallel

Shift complexity onto the compiler, who must find out about data dependencies