Drawbacks of single cycle implementation

A single cycle machine has disadvantages such as:

1. All instructions take the same time (CPI = 1) but some instructions in fact are shorter than others, for example:
   - an ADD uses, in order, Instruction memory, Register file, ALU, and Register file again
   - a LW uses Instruction memory, Register file, ALU, Data memory, and Register file.
2. Some combinational units must be replicated since used in the same cycle

Alternative to single cycle implementation

An alternative is to have a shorter cycle and have each instruction executing in multiple shorter cycles.

The shorter cycles will be determined by the delay of individual functional units.

We should be able to streamline the resources since we might be able to reuse the same unit at different cycles.

We will have to add some registers (not visible at the ISA level) since some information has to be kept from one cycle to the next.

Multiple cycle implementation

Break up the “instruction execution cycle” in a sequence of steps.

1. Instruction fetch
2. Instruction decode and source register(s) read
3. ALU execution
4. Memory access (read/write) or ALU completion (write result register)
5. Write back register in case of a Load instruction

Instruction fetch

Instruction fetch. We need an Instruction Register (IR) to hold the current instruction since the information in the instruction will be needed in subsequent cycles.

1. IR $\leftarrow$ Memory[PC]
2. PC $\leftarrow$ PC + 4

Resources At this point, there is no need to distinguish between Instruction and Data memory. We can also use the ALU to increment the PC.

We need to have an IR, a register in the processor but not one that is visible to the ISA.

Instruction decode and read source register(s)

Instruction decode and register fetch. Let A correspond to Input 1 of ALU and B correspond to Input 2.

1. A $\leftarrow$ Reg[IR[25:21]] (i.e., read rs)
2. B $\leftarrow$ Reg[IR[20:16]] (i.e., read rt)
3. ALUout $\leftarrow$ PC + (sign-extend(IR[15:0]*4) (just in case we had a branch)

Resources We need two non-ISA-visible registers A and B since there contents is needed for execution (in the next cycle).

Since the register fetch is not using the ALU, let's use the ALU to compute the target address of a branch, just in case we had a branch! The target address will be put in a non-ISA-visible register ALUout.

ALU execution (either arithmetic or address computation)

ALU execution or *Memory address computation* or *Branch address setting*

If the instruction is R-type

- ALUout $\leftarrow$ A op. B

If the instruction is Memory (Load-Store) or Immediate

- ALUout $\leftarrow$ A + sign-extend(IR[15:0])

If the instruction is Branch (and successful)

- If (A = B) then PC $\leftarrow$ ALUout (note this is the value of the ALUout computed in the previous cycle)

Resources The ALUout (non-ISA-visible register) is needed for passing the value of the computation to the next step.

Of course, the ALU is needed also!

Memory access or ALU completion (write result register)

Memory access or *ALU completion*

- If Load: Memory-data $\leftarrow$ Memory[ALUout]
- If Store: Memory[ALUout] $\leftarrow$ B
- If arithmetic: Reg[IR[15:11]] $\leftarrow$ ALUout

Resources Memory and a non-ISA-visible register Memory-data in case of a Load.

Load completion

- Reg[IR[20:16]] $\leftarrow$ Memory-data

Streamlining of Resources (cf. Figure 5.31)

We have "streamlined" the number of resources:

- No distinction between Instruction and Data Memory
- Only one ALU
- A few more MUXes (e.g., one is needed for the memory address) and temporary registers (e.g., IR) with their control.