# Single Cycle MIPS

## Add the J (Jump) instruction

See Fig 5.24

# Performance

Total time to execute benchmark programs is an important measure of the performance.

Total time = Cycles Per Instruction (CPI) x number of instructions executed x Cycle time

For the single cycle MIPS, CPI = 1 but each instruction takes a different amount of time.
Since no one has a crystal ball, one solution is to make the clock period as large as is needed for the execution of the slowest instruction. But this is not efficient.

# The Role of Compilers

## Machine-independent Optimization

```
x := y + z + P.w
.
.
y := z + P.w
```

The execution will be faster if the value of z + P.w that is evaluated in line 1 is saved in a register for future use. The identification of common sub-expressions helps improve performance.

## Machine-dependent Optimization

Choose the best possible machine instructions for a given program.

Rearrange instructions to improve pipeline performance:

| | |
|---|---|
| a := b + 1 | a := b + 1 |
| c := a + 2 | d := b + 3 |
| d := b + 3 | . . . . . . . . |
| . . . . . . . . | c := a + 2 |

## Register Allocation

For maximum speedup, avoid memory access and keep the variables in registers as long as possible.

This relies on how many registers are available:

**Example.** $F = (a+b)/(a-b) + c.d.(c-d)$

In evaluating **F** on a load-store architecture, what is the minimum number of registers required in the CPU, so that no variable has to be loaded or stored more than once?

```
Load r1, a
Load r2, b
Add r3, r2, r1        * r3 contains a+b
Sub r4, r2, r1        * r4 contains a-b
Div r3, r3, r4        *first part saved in r3*
```

```
Load r1, c
Load r2, d
Sub r3, r1, r2          * r3 contains c-d
Mul r1, r1, r2          * r1 contains c.d
Mul r4, r1, r3          *second part saved in r4*
Add r3, r3, r4
Store r3, F
```

We need at least four registers.

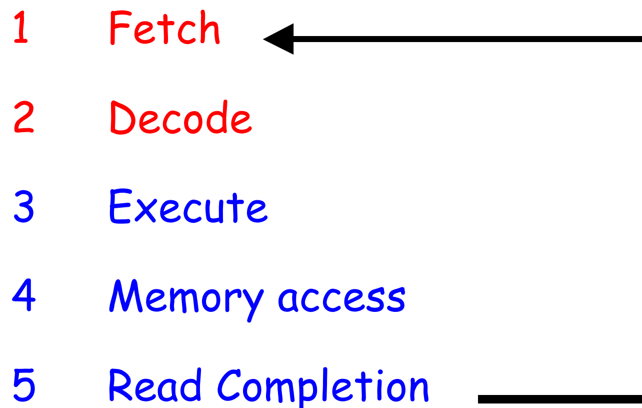
# Multi-cycle Implementation of MIPS

The cycle-time of the 1 CPI (Cycles Per Instruction) MIPS will be equal to the time required to complete the longest instruction. This will slow down the execution. This implementation, although quite simple, is not attractive.

In the multi-cycle implementation, each instruction takes a few cycles. The number of cycles is different for different instructions. Note the various buffers between the stages. Why are they added?

See figures 5.25-5.28 (Fig 5.27 has a mistake: MDR output should be connected to the 1 input of the MUX)

# Instruction Cycle for Multi-cycle MIPS




The first two steps are the same for all instructions. The last three steps are different for different instructions. Fig. 5.30 contains a summary. The details will follow:

# Execution of MIPS instructions

**1. Instruction Fetch (F)**

IR:= M[PC] — fetch the instruction

PC:= PC+4 — calculate next PC

**2. Instruction Decode and Register Fetch (D)**

A:= R[$IR_{25..21}$] — fetch RegFile[rs] into A

B:= R[$IR_{20..16}$] — fetch RegFile[rt] into B

ALUOut:= PC + $IR_{15..0}$ — Compute branch target address

and of course, decode the instruction.

**3. Execute/effective address/branch completion (X)**

| | |
|---|---|
| 3.1 Load or Store | ALUout := A + $IR_{15..0}$ |
| 3.2 R-R ALU operation | ALUout := A op B |
| 3.3 Branch | if A=B then PC := ALUout |
| 3.4 Jump | PC:= PC[31:28]. 4*IR[25..0] |

## 4. Memory access / R-type Completion (M)

**4.1 Load** MDR:= M[ALUout]

**4.2 Store** M[ALUout]:= B

**4.3 R-type Completion** RegFile[IR15:11] := ALUout

## 5. Memory read completion (Write Back) (W)

**Load** RegFile[$IR_{20..16}$]:= MDR

No activity for branch or store operations or R-type instructions here.