Pipelines (Also refer to Professor Roumani's slides for on this.)

The Timing of Single-cycle vs. Multi-cycle

A standard that studies various popular programs indicates that in a typical program:

- 20% of instructions are branches
- 10% of instructions are jumps
- 15% of instructions are load/store (approximately 10% load)
- 55% of instructions are R-types

In a typical CPU we have the following latencies:

- IF = DM = 2ns
- ALU = 2ns
- RF = 1.5ns

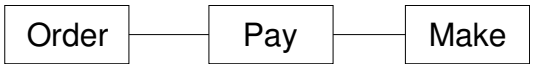
With these data a program with 100 instructions will take approximately:

- 900 ns in a single-cycle CPU
- 760 ns in a multi-cycle CPU

Of course the multi-cycle design is noticeably faster, but is there a way to make the CPU even faster?

Latency vs. Throughput

Assume you are in small fast food restaurant with three employees, each one doing a single task in 1 minute.



It will take every person 3 minutes to enter the store and leave with the food.

Now they let another customer in for order while the previous customer is at Pay, then we will see one customer leave the restaurant every minute. Of course still it take 3 minutes for a single customer to receive the order, but now three times the number of customers are served.

This is the difference between latency and throughput.

We use the same concept in a pipelined CPU:

Again, we are assuming each part has a latency of 2 ns.

The latency of this design is 10 ns which is higher than S/C (= 9 ns) and M/C (= 6-10 ns), however, for a program with 100 instructions the total time is only approximately 200 ns.
Note that this design has a high hardware redundancy but since no device is going to be idle it will not be a problem.

Hazards

Unfortunately this design has a few hazards:

- **Structural:** What if an instruction needs to go back and use a part again?
  There is no solution to this problem.
  Fortunately since MIPS is designed for pipelining this will not happen.
- **Data Dependency:** What if an instruction needs the data from a previous instruction?
  For example the following small program:

```
addi $t0, $s0, 1
add  $a0, $t0, $a0
```

  This must be avoided!
- **Branch:** We do not know if the two instructions after branch are to be executed unless we know the result of the branch. This should also be avoided:
  - One solution is to let those instructions run and kill them later:
    Make them harmless by setting: RegWrite = MemWrite = MemRead = 0
    But this will take approximately double the time.
  - A better solution is to take the 2 statements before the branch and execute them after branch.

We'll get back to hazards later.

Implementation

– We use the same subset we used for single/multi-cycle to implement the pipelined data path.
– We have to balance the speed so all instructions move at the same speed.
– This increases latency, however we still have a higher put through.

First we implement the data path assuming there are no hazards:

See the pipelined data path in Professor Roumani's notes or in the textbook. (fig 6.30)

Note that this diagram is derived from the single-cycle data path, except we have added registers to keep track of the data and results from previous steps for each instruction.

Also note that these registers are pretty large and have separate sections:

– These parts are shown bottom-up in the diagram.
– Italic/Gray text means a copy of the same data taken from previous register.

Register	***IF/ID***	***ID/EX***	***EX/Mem***	***Mem/WB***
Different Parts (and # of bits)	PC + 4 (32) IR (32)	DstReg (5 + 5) Immediate (32) RFout (32 x 2) *PC + 4 (32)* EX (4) M (3) WB (2)	RegDst (5) *RFlower (32)* ALUout (32) Zero (1) BranchAdd (32) *M (3)* *WB (2)*	*RegDst (5)* *ALUout (32)* lw (32) *WB (2)*
Total # of bits	64	147	107	71

Back to Hazards

Now let's take care of hazards we mentioned in our new data path.

We only talk about hazard qualitatively.

- **Branch Hazards:** How do we know if you should even execute the instructions after branch without knowing the result of the branch?
  - 1. Play it safe. Introduce three stalls (bubble). i.e. trash three times after branch.
    - Use a safe instruction
      E.g. `sll $0, $0, 0` (This is the same as pseudo instruction `nop`)
      If you look at this instruction in machine language it is a bunch of zeros!
    - OR just set MemRead = MemWrite = RegWrite = 0
      It is not necessary to set MemRead = 0 but it will avoid bothering the bus.
    - Assuming a program with 100 instructions and no hazards takes 200ns with this method it will take approximately 320ns.
    - Macintosh used to use this method in their older systems.
  - 2. Assume Branch not taken, then trash if taken
    - But why waste 3 instructions for every branch? The previous method was a lose/lose situation, while this is a 50/50 win/lose situation.
    - With this method the program mentioned earlier would take between 200-320ns.
  - 3. Speculative (or Adaptive)
    - Once the branch is called store the result somewhere.
    - If the branch was taken last time, assume it is taken.
    - Great for loops.
    - One of the best method that is being used today.

    - With this method the program mentioned earlier would take approximately 200+0.2x20 or 204ns.
- 4. MIPS Solution (aka. Delayed)
    - This is perhaps the most complex method.
    - It involves a combination of both hardware and software.
    - We can find out the result of the branch sooner by looking at the “zero” output of the ALU. In face we can move that ALU to a previous step which makes us two instructions ahead.
    - The instruction after the branch (that will be executed either way) must be an independent instruction.
        - This instruction is chosen by the software.
        - A loop in a high level language consists of many instructions and there is usually an independent instruction that we can use.
    - With this method the program mentioned earlier would take approximately 200+0.2x20 or 204ns. Just like the previous method.

Fig 6.30

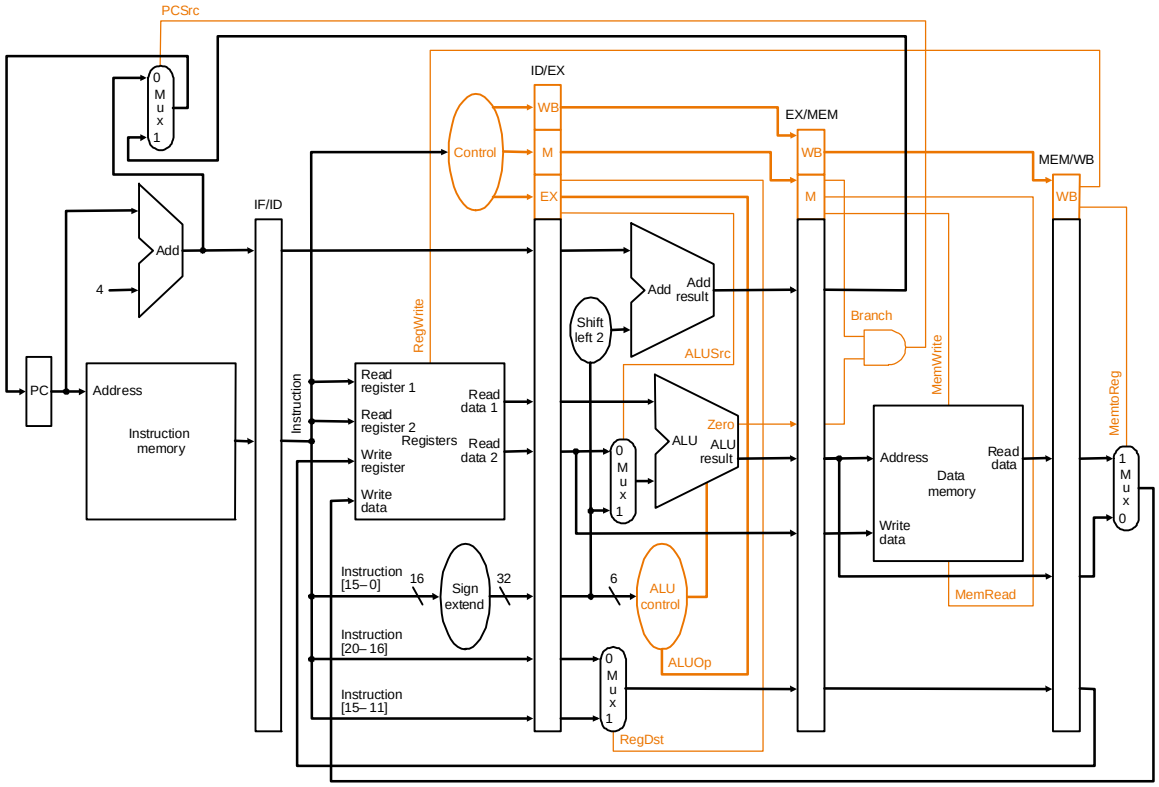
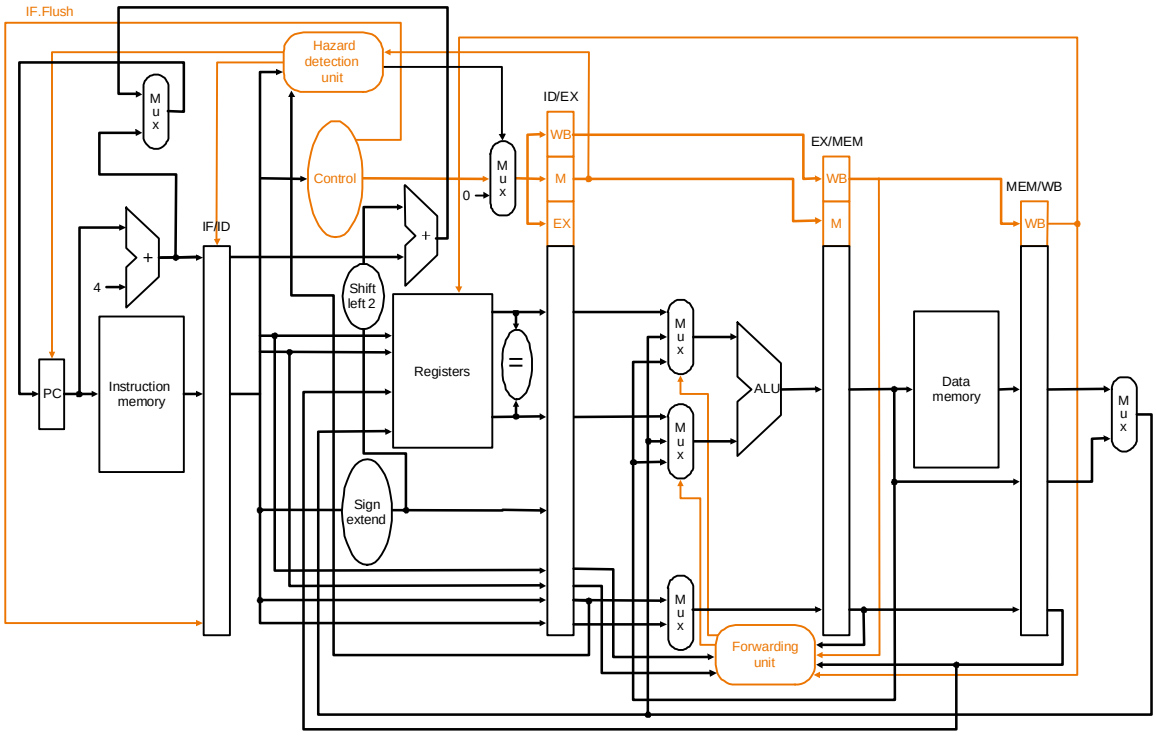

PCSrc
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
Add
4
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
RegWrite
Shift left 2
Add result
ALUSrc
Zero
ALU
ALU result
Branch
MemWrite
Address
Data memory
Read data
Write data
MemRead
MemtoReg
Instruction [15–0]
16
Sign extend
32
6
ALU control
ALUOp
Instruction [20–16]
Instruction [15–11]
RegDst
0
M
u
x
1

Fig 6.51


IF.Flush
Hazard detection unit
M u x
Control
0
M u x
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
IF/ID
+
4
+
Shift left 2
PC
Instruction memory
Registers
=
Sign extend
M u x
M u x
M u x
ALU
Data memory
M u x
Forwarding unit