

3. Shift the Divisor register right 1 bit
 <33re rep -> repeat
 Test remainder
 <0
 Add divisor register to remainder and place in Remainder register

Example:

Perform $n + 1$ iterations for n bits

Remainder 0000 1011

Divisor 0011 0000

Iteration 1:

(subtract)

Rem 1101 1011

Quotient 0

Divisor 0001 1000

Iteration 2:

(add)

Rem 1111 0011

Q 00

Divisor 0000 1100

Iteration 3:

(add)

Rem 1111 1111

Q 000

Divisor 0000 0110

Iteration 4:

(add)

Rem 0000 0101

Q 0001

Divisor 0000 0011

Iteration 5:

(subtract)

Rem 0000 0010

Q 0001 1

Divisor 0000 0001

Since remainder is positive, done.

Q = 0011 and Rem = 0010

3.42(10)

a.

$$\begin{array}{r} \text{Convert } +1.1011 \times 2^{14} + -1.11 \times 2^{-2} \\ 1.1011\ 0000\ 0000\ 0000\ 0000\ 000 \\ -0.0000\ 0000\ 0000\ 0001\ 1100\ 000 \\ \hline 1.1010\ 1111\ 1111\ 1110\ 0100\ 000 \\ 0100\ 0110\ 1101\ 0111\ 1111\ 1111\ 0010\ 0000 \end{array}$$

b. Calculate new exponent:

$$\begin{array}{r} 111\ 11\ 1 \\ 100\ 0110\ 1 \\ +011\ 1110\ 1 \\ \hline 1000\ 0101\ 0 \\ -011\ 1111\ 1 \quad \text{minus bias} \\ \hline 1111\ 1111 \\ \hline 100\ 0101\ 1 \quad \text{new exponent} \end{array}$$

Multiply significands:

$$\begin{array}{r} 1.101\ 1000\ 0000\ 0000\ 0000\ 0000 \\ \times 1.110\ 0000\ 0000\ 0000\ 0000\ 0000 \\ \hline 1\ 11\ 11 \\ 1\ 1011\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 \\ 11\ 0110\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 \\ +1.10\ 1100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 \\ \hline 10.11\ 1101\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 \end{array}$$

Normalize and round:

exponent 100 0110 0

significand

1.011 1010 0000 0000 0000 0000

Signs differ, so result is negative:

1100 0110 0011 1010 0000 0000 0000 0000

3.44(10)

a. $2^{15} - 1 = 32767$

b.

$$\begin{aligned} 2.0_{\text{ten}} \times 2^{15} \\ 2^{21} &= 3.23 \times 10^{616} \\ 2^{22} &= 1.04 \times 10^{1233} \\ 2^{23} &= 1.09 \times 10^{2466} \\ 2^{24} &= 1.19 \times 10^{4932} \\ 2^{25} &= 1.42 \times 10^{9864} \end{aligned}$$

so

as small as $2.0_{\text{ten}} \cdot 10^{-9864}$

and almost as large as $2.0_{\text{ten}} \cdot 10^{9864}$

c. 20% more significant digits, and 9556 orders of magnitude more flexibility.
(Exponent is 32 times larger.)

5.1(5)

Combinational logic only: a, b, c, h, i

Sequential logic only: f, g, j

Mixed sequential and combinational: d, e, k

5.3(10)

a. RegWrite = 1: `sw` and `beq` should not write results to the register file. `sw`(`beq`) will overwrite a random register with either the store address (branch target) or random data from the memory data read port.

b. ALUOp0 = 1: `lw` and `sw` will not work correctly because they will perform subtraction instead of the addition necessary for address calculation.

c. ALUOp1 = 1: `lw` and `sw` will not work correctly. `lw` and `sw` will perform a random operation depending on the least significant bits of the address field instead of addition operation necessary for address calculation.

d. Branch = 1: Instructions other than branches (`beq`) will not work correctly if the ALU Zero signal is raised. An R-format instruction that produces zero output will branch to a random address determined by its least significant 16 bits.

e. MemRead = 1: All instructions will work correctly. (Data memory is always read, but memory data is never written to the register file except in the case of `lw`.)

f. MemWrite = 1: Only `sw` will work correctly. The rest of instructions will store their results in the data memory, while they should not.

5.11(10)

A modification is required for the datapath of Figure 5.17 to perform the autoincrement by adding 4 to the `$rs` register through an incrementer. Also we need a second write port to the register file because two register writes are required for this instruction. The new write port will be controlled by a new signal, "Write 2", and a data port, "Write data 2." We assume that the Write register 2 identifier is always the same as Read register 1 (`$rs`). This way "Write 2" indicates that there is second write to register file to the register identified by "Read register 1," and the data is fed through Write data 2.

A new line should be added to the truth table in Figure 5.18 for the `linc` command as follows:

RegDst = 0: First write to `$rt`.

ALUSrc = 1: Address field for address calculation.

MemtoReg = 1: Write loaded data from memory.

RegWrite = 1: Write loaded data into `$rt`.

MemRead = 1: Data memory read.

MemWrite = 0: No memory write required.

Branch = 0: Not a branch, output from the PCSrc controlled mux ignored.

ALUOp = 00: Address calculation.

Write2 = 1: Second register write (to `$rs`).

Such a modification of the register file architecture may not be required for a multiple-cycle implementation, since multiple writes to the same port can occur on different cycles.

5.14(10)

swap \$rs,\$rt can be implemented by

```
addi $rd,$rs,0
```

```
addi $rs,$rt,0
```

```
addi $rt,$rd,0
```

if there is an available register \$rd

or

```
sw $rs,temp($r0)
```

```
addi $rs,$rt,0
```

```
lw $rt,temp($r0)
```

if not.

Software takes three cycles, and hardware takes one cycle. Assume R_s is the ratio of swaps in the code mix and that the base CPI is 1:

$$\text{Average MIPS time per instruction} = R_s * 3 * T + (1 - R_s) * 1 * T = (2R_s + 1) * T$$

$$\text{Complex implementation time} = 1.1 * T$$

If swap instructions are greater than 5% of the instruction mix, then a hardware implementation would be preferable.

5.32(10)

We use the same datapath, so the immediate field shift will be done inside the ALU.

1. Instruction fetch step: This is the same ($IR \leftarrow \text{Memory}[PC]$; $PC \leftarrow PC + 4$)

2. Instruction decode step: We don't really need to read any register in this stage if we know that the instruction in hand is a lui, but we will not know this before the end of this cycle. It is tempting to read the immediate field into the ALU to start shifting next cycle, but we don't yet know what the instruction is. So we have to perform the same way as the standard machine does.

$$A \leftarrow 0 (\$r0); B \leftarrow \$rt; \text{ALUOut} \leftarrow PC + (\text{sign-extend}(\text{immediate field}));$$

3. Execution: Only now we know that we have a lui. We have to use the ALU to shift left the low-order 16 bits of input 2 of the multiplexor. (The sign extension is useless, and sign bits will be flushed out during the shift process.)

$$\text{ALUOut} \leftarrow \{\text{IR}[15-0], 16(0)\}$$

4. Instruction completion: $\text{Reg}[\text{IR}[20-16]] = \text{ALUOut}$.

The first two cycles are identical to the FSM of Figure 5.38. By the end of the second cycle the FSM will recognize the opcode. We add the Op='lui', a new transition condition from state 1 to a new state 10. In this state we perform the left shifting of the immediate field: $\text{ALUSrcA} = x$, $\text{ALUSrcB} = 10$, $\text{ALUOp} = 11$ (assume this means left shift of ALUSrcB). State 10 corresponds to cycle 3. Cycle 4 will be translated into a new state 11, in which $\text{RegDst} = 0$, RegWrite , $\text{MemtoReg} = 0$. State 11 will make the transition back to state 0 after completion.

As shown above the instruction execution takes 4 cycles.

5.36(5)

Effective CPI = $\text{Sum}(\text{operation frequency} * \text{operation latency})$

MIPS = $\text{Frequency}/\text{CPI}_{\text{effective}}$

Instruction	Frequency	M1	M2	M3
Loads CPI	25%	5	4	3
Stores CPI	13%	4	4	3
R-type CPI	47%	4	3	3
Branch/jmp CPI	15%	3	3	3
Effective CPI		4.1	3.38	3
MIPS		976	946	933

From the results above, the penalty imposed on frequency (for all instructions) exceeds the gains attained through the CPI reduction. M1 is the fastest machine.

The more the load instructions in the instruction mix, the more the CPI gain we can get for the M2 and M3 machines. In the extreme case we have all instructions loads, M1 MIPS = 800, M2 MIPS = 300, and M3 MIPS = 933.3, so M3 becomes the best machine in such a case.

5.37(10)

Effective CPI = $\text{Sum}(\text{operation frequency} * \text{operation latency})$

MIPS = $\text{Frequency}/\text{CPI}_{\text{effective}}$

Instruction	Frequency	2.8 GHz CPI	5.6 GHz CPI	6.4 GHz CPI
Loads CPI	26%	5	6	7
Stores CPI	10%	4	5	6
R-type CPI	49%	4	4	5
Branch/jmp CPI	15%	3	3	4
Effective CPI		4.11	4.47	5.47
MIPS		1167.9	1250	1170.0

The two-cycle implementation increases the frequency, which benefits all instructions, and penalizes only loads and stores. The performance improvement is 7% relative to the original implementation.

Further increase of the clock frequency by increasing the instruction fetch time into two cycles will penalize all instructions and will reduce the performance to about the same as that of the 4.8 GHz base performance. Such implementation hurts the CPI more than the gain it brings through frequency increase and should not be implemented.