# Final Exam Study Guide

Revision of 12 December 2016

What you may bring:

- This document
- A pen or pencil
- A calculator to converts between decimal, hex and binary
- A sheet of paper with hints and tips – printed or hand-written, double-sided, 8.5x11”
- A bottle of water

What you should not bring:

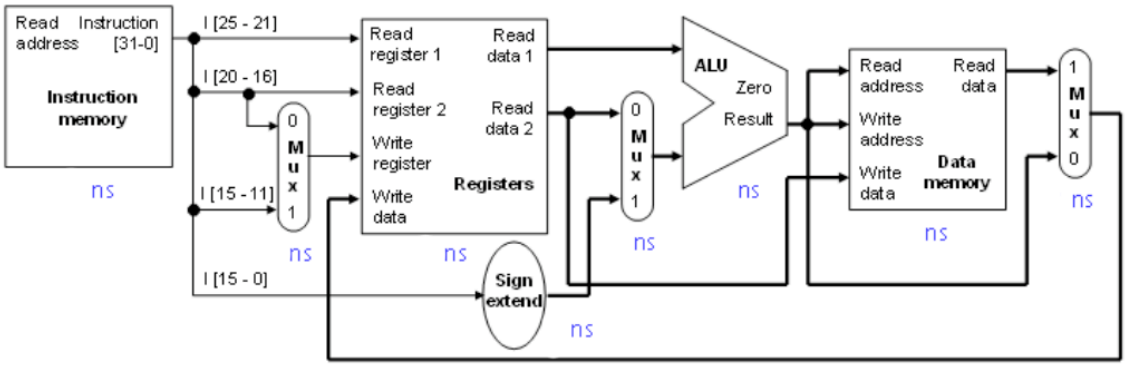
- Messy or noisy food – candy is fine if it doesn’t have noisy wrappers

What I will provide:

- A MIPS green card, KiB to KB conversion table

Themes to study:

1) Single-path processors.
   You will be given a single-stage processor diagram like the following:




Note that the diagram excludes all the program counter elements.

It will have numeric values filled in where just “ns” is shown in blue now to indicate how many nanoseconds each element requires to process its part of an instruction.

You will be given a table that looks like this:

| Instruction | Instr Mem | Reg Mux | Reg Mem | Sign Ext | ALU Mux | ALU | Data Mem | Write Mux | Total Time |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

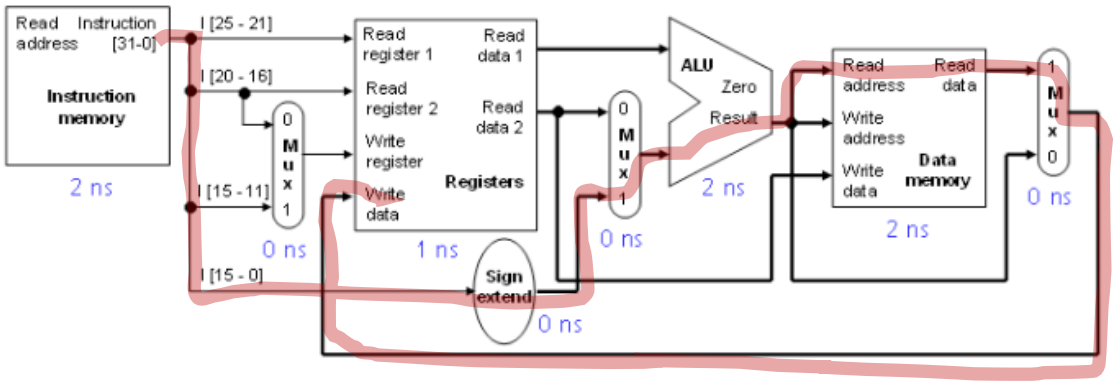
For a given list of instructions, you will be asked to:

- Indicate which of the elements are used by marking an "X", and
- Calculate the total time it takes for the instruction to process completely and enter it in the last column

You will place an "x" in the field if the instruction given uses that element, and leave it blank if it does not use that element. For the last column you will specify the total amount of time taken.

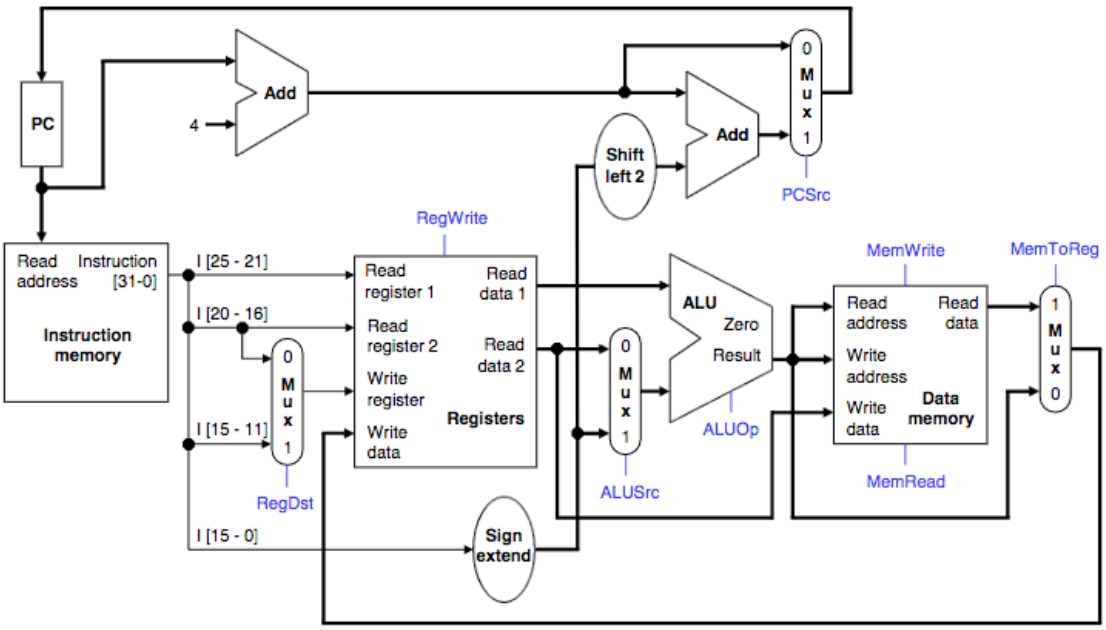
- Tracing an instruction
  You will be asked to trace an instruction through the processor. Instructions that use more than one input (like R-type instructions) take slightly different paths simultaneously. You will be asked to take a path that passes through the fewest or greatest number of elements by following the paths provided in the diagram.

  For example, a trace of
  lw $rt, address would look like this:




  and the total time would be 2+0+0+2+2+0 = 6ns through 6 elements.

2) Control Unit.
   Based on problem 4.1. Be able to describe which control signals are used for an instruction. We will use this diagram:




This diagram has all the same control signals as the diagram we used in our review lecture, with one exception: we have simplified the PCSrc signal. Technically, the ALU's zero signal is AND'd with PCSrc but that has been eliminated to make the diagram clearer.

You will be given a series of MIPS commands and the table below, and asked to determine which of six control signals are used and asked to specify what math operation the ALU will perform. All commands use the program counter, but only some use memory. Most but not all write to registers. And some don't use the ALU.

If the field is used, you will be asked to specify its value as 1 or 0 except for the ALU field where you will need to provide the math function – add, subtract, multiply, divide, AND, OR, XOR or NOT. Note that the PC Src column is intentionally excluded.

| Operation | Reg Dst | Reg Write | ALU Src | ALU Op | Mem Write | Mem Read | Mem to Reg |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

This table may help:

| Signal name | Effect when deasserted (set to "0") | Effect when asserted (set to "1") |
|---|---|---|
| **RegDst (RD)** | The register destination number for the Write register comes from the rt field (bits 20:16) | The register destination number for the Write register comes from the rd field (bits 15:11). |
| **RegWrite (RW)** | None. | The register on the Write register input is written with the value on the Write data input. |
| **ALUSrc (AS)** | The second ALU operand comes from the second register file output (Read data 2). | The second ALU operand is the sign-extended, lower 16 bits of the instruction. |
| ALU | A signal to tell the ALU what operation to perform. Use a word here like "add" or "subtract" or "check if equal", not 1s or 0s. | |
| **PCSrc (PCS)** | The PC is replaced by the output of the adder that computes the value of PC+4. | The PC is replaced by the output of the adder that computes the branch target. |
| **MemRead (MR)** | None. | Data memory contents designated by the address input are put on the Read data output. |
| **MemWrite (MW)** | None. | Data memory contents designated by the address input are replaced by the value on the Write data input. |
| **MemtoReg (MTR)** | The value fed to the register Write data input comes from the ALU. | The value fed to the register Write data input comes from the data memory. |

Non-pipelined processor

Be familiar with what elements in the processor each type of instruction uses. We won't have any instructions you have not seen before, but you will need to be able to mark which CPU elements an instruction uses. For example, we know store word (sw) uses the main memory, and that add immediate unsigned (addiu) uses the sign extender. You will be given a table with instructions based on the single-stage processor and asked to indicate which elements are used. You will mark an "x" in each element that is required to process the instruction.

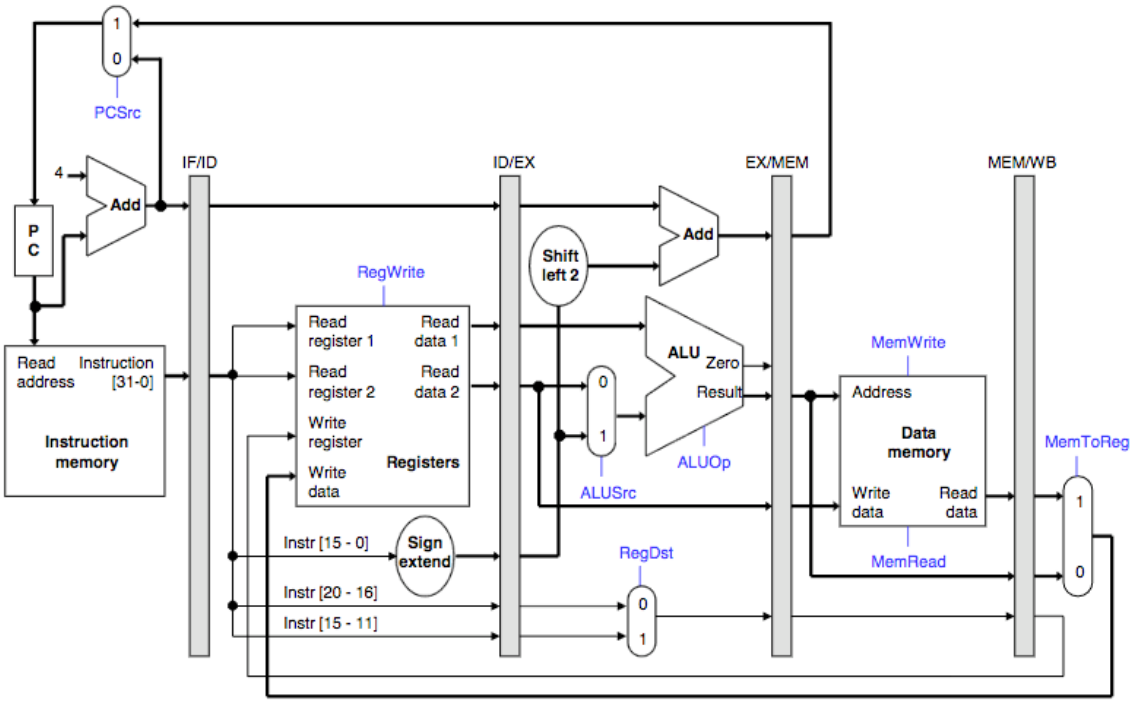
| Instruction | Instr Mem | Reg Mux | Reg Mem | Sign Ext | ALU Mux | ALU | Data Mem | Write Mux |
|---|---|---|---|---|---|---|---|---|
| `Xor $rd, $rs, $rt` | x | x | x | | x | x | | x |

In this example, because xor is an r-type instruction, it uses the elements marked.

Control signals

All control signals are determined when the instruction is decoded. In a pipelined design, we divide up the elements into stages, and use special pipeline registers between the stages to carry signals forward to the stage where they're needed.

For MIPS, we can move around the elements a bit, and simplify the main memory unit to only take in one address.
Here is a diagram showing these four registers added:

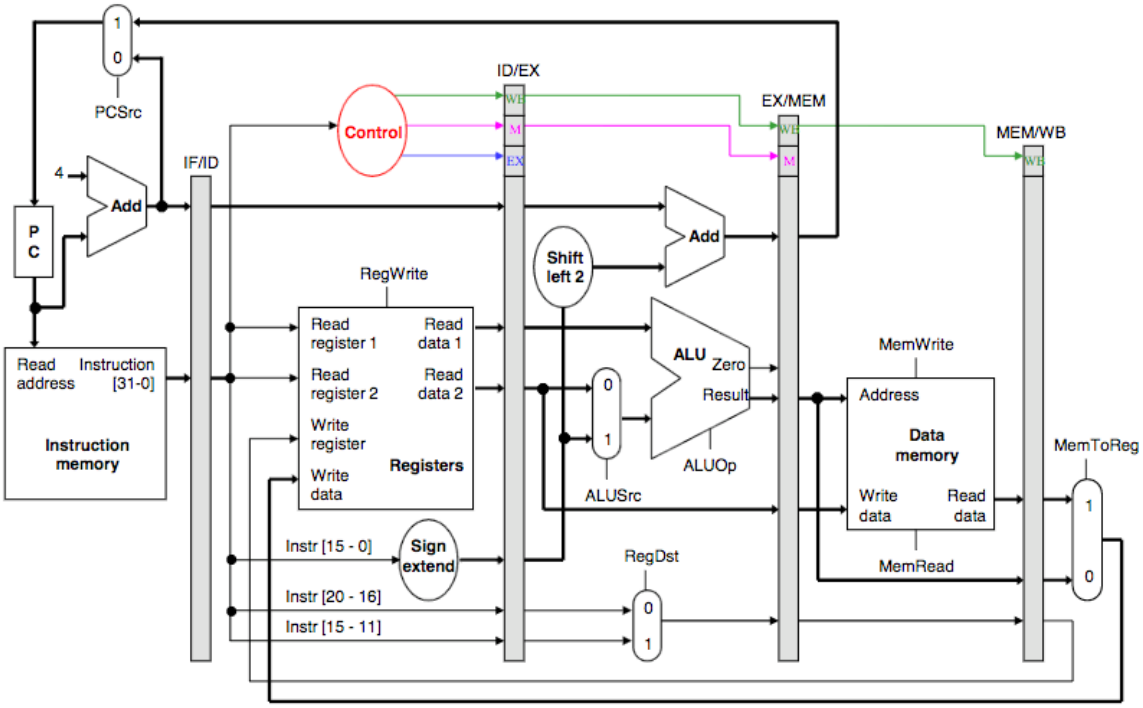



Each of the control signals are in blue. Some versions have more or fewer control signals. For this one, know that the control signals are determined in the second stage, and then delivered to the required stage by passing through the buffers.

The pipeline buffers exist both to carry values forward from the normal elements like the ALU and memory, as well as to carry forward control signals.
In normal operation, each instruction passes through all five stages and carries along with it control signals to be used in later stages.

Q: You will need to answer questions about **when** each signal is used that will require thinking about the path the signal takes to reach its target – does it go through buffers? Which ones?

This diagram may be helpful. It gives the general idea that although all control signals are created in the instruction decode stage, signals for the execution stage are carried forward through one set of pipeline buffers, control signals for the memory stage are carried forward two stages and so on.




Q: You will be given a short MIPS program and asked to specify which instructions are in the pipeline and in which stage after a certain number of cycles.

- Based on problem 4.3: processor improvements. I will clearly state if alternate components are additions or replacements.

I've updated the table a bit to look like this:

| | Fetch | Decode | Execute | Memory | Writeback |
|---|---|---|---|---|---|
| a. | 100ps | 150ps | 125ps | 400ps | 200ps |
| b. | 150ps | 80ps | 200ps | 180ps | 200ps |

You'll be given values in picoseconds for each stage, and asked to compare the two CPUs. Know how to calculate the cycle time for a single-stage (non-pipelined) processor versus one that is pipelined. Know how to reduce the cycle time by breaking your slowest component into multiple pieces. Think about when this would work to speed things up and when it would not.

Expect questions on clock cycle time and speedup, and the cost/performance ratio.

- Based on problem 4.8, pipelining. Be open to the possibility of more or fewer pipeline stages. Be able to compare single-cycle and pipeline latencies.

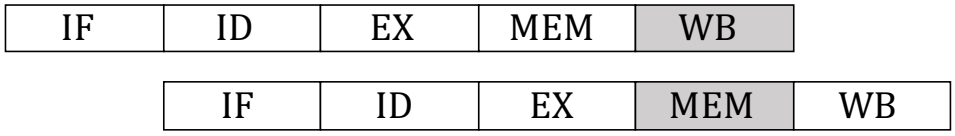
  Be able to spot dependencies and circle them.

- Know what RAW, WAR and WAW dependencies are and how to classify a dependency as one of the three.

- Know the difference between a dependency and a hazard. A dependence exists between two instructions, the latter of which depends on the value from the earlier.

  The hazard is the situation in the actual pipeline where in a given cycle, the value being generated by the first instruction is not available yet and the second instruction needs it now. For example, a value may not be generated until after the write-back stage of a first instruction, by which time we're 2 cycles past when the next instruction depending on it needed it. In this example, the first instruction has a value available in WriteBack that the second instruction needs in its Memory Stage. So we have shaded those two boxes.

  You will be asked to circle or shade two stages similarly given a series of instructions.




- Be able to convert a single-stage diagram to a pipelined diagram:

  Update: I will give you the values of the various stages and ask you to calculate the program execution time based on a non-pipelined and a pipelined arrangement.

- Be able to draw forwarding on a graphical diagram such as this:

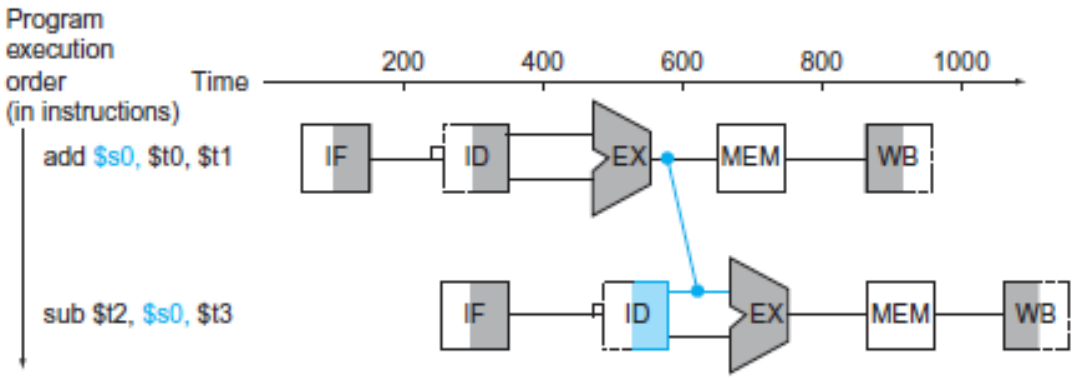
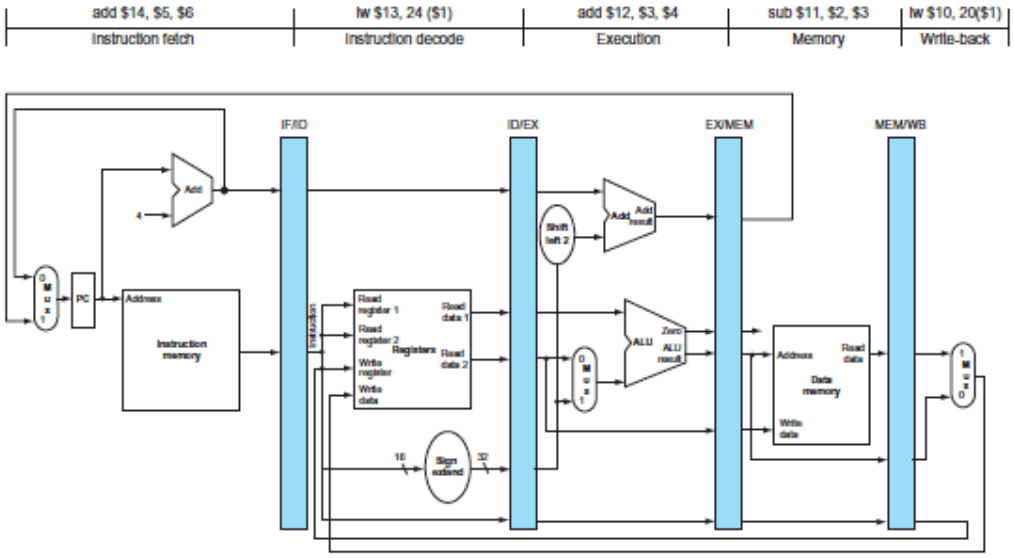



**FIGURE 4.29 Graphical representation of forwarding.** The connection shows the forwarding path

At a particular time, be able to show where instructions are in a pipeline and where variables sit in buffers.




- Hamming Codes

We will have a question on Hamming Codes
You will be given a table like this

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Data Given | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | | | |
| | | | | | | | | | | | | |
| | data bit position | | | 1 | | 2 | 3 | 4 | | 5 | 6 | 7 | 8 |
| | data bit value | | | 0 | | 1 | 1 | 1 | | 1 | 0 | 0 | 0 |
| | bit position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| | | p1 | p2 | d1 | p4 | d2 | d3 | d4 | p8 | d5 | d6 | d7 | d8 |
| Coverage Matrix | p1 | x | | x | | x | | x | | x | | x | |
| | p2 | | x | x | | | x | x | | | x | x | |
| | p4 | | | | x | x | x | x | | | | | x |
| | p8 | | | | | | | | x | x | x | x | x |
| | p16 | | | | | | | | | | | | |
| | p32 | | | | | | | | | | | | |
| | p64 | | | | | | | | | | | | |

It shows the 8 bits of data given and how they lay out across the matrix.
Your job will be to calculate the value of P1, P2, P4, and P8. You can ignore the other rows (p16, p32, p64) – they come from a tool I built for longer strings.

Taking the first row, because there are "x"s in columns p1, d1, d2, d4, d5, and d7, we start by writing down the digits we have. We don't yet have a value for p1, so we'll ignore it. That leaves us with 0, 1, 1, 1, and 0. There are an odd number of 1s there, so we'll make p1 = 1 to make it even. Always make it even. The other way to calculate this is to just XOR the bits. That does **exactly** the same thing. XOR a string

with an odd # of 1's and you'll get a 1. XOR a string with an even # of 1s and you'll get a 0. For now it's just as easy to look and write it down, but in case XOR was confusing to you, it's really easy.

Once we have the values, we add them to the string and write out the 12-bit result:

| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|

The blue bits are the parity bits we added.

The second half of the exercise will be to take an assigned 12-bit string and figure out which p bits are right and which are wrong. You'll be given the 12 bits and the table to complete. If you XOR all the bits for a row INCLUDING the parity bit you are given, it should always be 0. If it isn't, it's an error.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bits Received | | | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| | Bit Position | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| | | | | | | | | | | | | | | |
| data bit position | | | | | 1 | | 2 | 3 | 4 | | 5 | 6 | 7 | 8 |
| data bit value | | | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| | | | | | | | | | | | | | | |
| | bit position | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| | | XOR | p1 | p2 | d1 | p4 | d2 | d3 | d4 | p8 | d5 | d6 | d7 | d8 |
| Re-calculated Bits | p1 | 0 | 0 | | 1 | | 0 | | 1 | | 0 | | 0 | |
| | p2 | 0 | | 0 | 1 | | | 1 | 1 | | | 1 | 0 | |
| | p4 | 0 | | | | 1 | 0 | 1 | 1 | | | | | 1 |
| | p8 | 1 | | | | | | | | 1 | 0 | 1 | 0 | 1 |
| | p16 | 0 | | | | | | | | | | | | |
| | p32 | 0 | | | | | | | | | | | | |
| | p64 | 0 | | | | | | | | | | | | |
| | | | | | | | | | | | | | | |
| Bit in error | | 8 | | | | | | | | | | | | |
| Corrected String | | | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |

So in our case, p1 covers 0⊕1⊕0⊕1⊕0⊕0 = 0, no error.
What about p2? 0⊕1⊕1⊕1⊕1⊕0 = 0, no error.
What about p4? 1⊕0⊕1⊕1⊕1 = 0, no error.
What about p8? 1⊕0⊕1⊕0⊕1 = 1. Error!!
We add up the positions of the parity bits in error – just 8 in this case, so bit 8 is in error.
We can just flip that bit and fix the problem:
0 0 1 1 0 1 1 1 0 1 0 1 becomes
0 0 1 1 0 1 1 0 0 1 0 1.

If we end up with p4, p2 and p1 in error, we just add up 4+2+1 and it tells us bit 7 is wrong. We can then correct that bit. And so forth.

Our last section covers caches. You will be given a series of addresses and a table similar to the following and asked to record the final entries in the table. You'll have to keep track of intermediate entries that are replaced, and count the number of compulsory, capacity and conflict misses, so you'll need to know the difference between them. Refer to section 5.8, page 459 of the text. Be prepared to go a second round (another round of the same addresses following the first), and to change the arrangement of the cache to 2- or 3- way.
For this example, the sequence of addresses was: 30, 11, 26, 35, 28, 50, 10, 47, 46, 51, 17, 40, 37, 9, 49, 13, 32, 44, 20 an 31.

| Set # (Index) | Word Address (in decimal) | Tag (in binary) | ¿Is the final entry in this cache the result of a: | | | | |
|---|---|---|---|---|---|---|---|
| | | | Compulsory Miss | Capacity Miss | Conflict Miss | Cache Hit | Unused |
| **0** | 32 | 0000 0010 | x | | | | |
| **1** | 49 | 0000 0011 | | | x | | |
| **2** | 50 | 0000 0011 | x | | | | |
| **3** | 51 | 0000 0011 | | | x | | |
| **4** | 20 | 0000 0001 | x | | | | |
| **5** | 37 | 0000 0010 | x | | | | |
| **6** | | | | | | | x |
| **7** | | | | | | | x |
| **8** | 48 | 0000 0010 | x | | | | |
| **9** | 9 | 0000 0000 | x | | | | |
| **10** | 10 | 0000 0000 | | | x | | |
| **11** | 11 | 0000 0000 | x | | | | |
| **12** | 44 | 0000 0010 | | | x | | |
| **13** | 13 | 0000 0000 | x | | | | |
| **14** | 46 | 0000 0010 | | | x | | |
| **15** | 31 | 0000 0001 | | | x | | |

Complete the statistics for this sequence:

| | | | |
|---|---|---|---|
| # of Conflict Misses | 6 | Total Cache Queries: | 20 |
| # of Compulsory Misses: | 14 | Total Misses: | 20 |
| # of Capacity Misses: | 0 | Hit Rate: | 0% |