Course pack for

EECS 373: Design of Microprocessor Based Systems

Fall semester 2002
v0.2

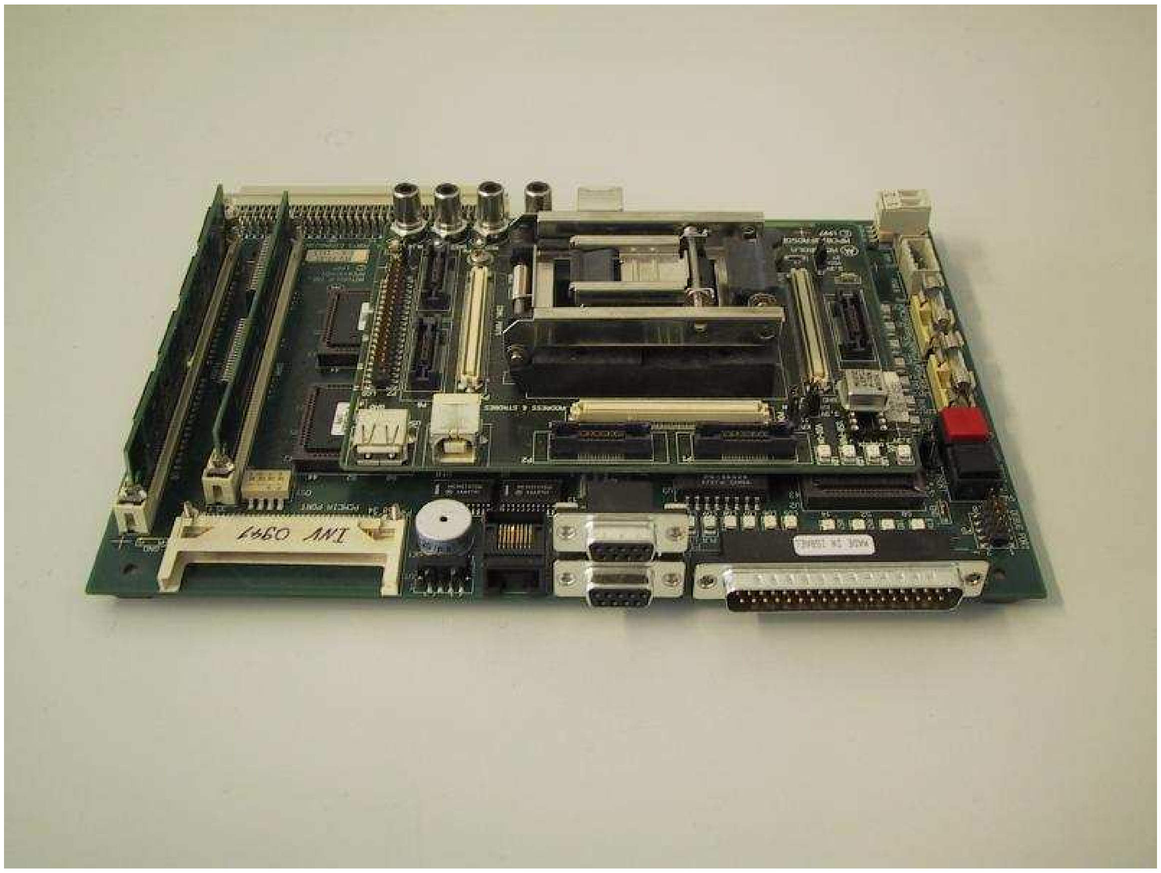


Table of Contents

I.About this document 3

  Disclaimer 3

  Purpose 3

  Text 3

II.Course Introduction 4

  Embedded systems at the Univ. of Michigan 4

  Other Introductory issues 5

III.PowerPC Assembly and Architecture 6

  Databook notation 6

  PowerPC Assembly Example 7

  PowerPC instructions 7

    Loads and Stores 8

    ALU instructions – not exactly orthogonal 8

    Other instructions 10

  PowerPC Architectural issues 10

    Special Purpose Registers 10

  Questions 11

IV.Memory Mapped I/O – the basics 13

  Simple memory-mapped I/O example 14

  Questions 14

V.Processor Bus Protocols 15

  Memory mapped I/O – again 15

VI.ABI and the Stack 15

VII.Interrupts 15

VIII.DMA – Direct Memory Access 15

IX.Counters and Timers 15

X.Analog and Digital Signal 15

XI.Bonus Topics 16

  Memory types 16

  Error correction 16

    Parity 16

    Error Correction Codes (ECC) 16

    Viterbi algorithm for error correction 16

  Software engineering in embedded systems 19

XII. 20

XIII. 20

I. About this document

Disclaimer

This document is not a purely original work! The basic ideas for this document came from previous EECS 373 offerings at the Univ. of Michigan. Work from Prof. Steve Reinhart, and Prof. Marios Papaefthymiou largely defined the course. Large parts of the document have been taken from other sources, sometimes verbatim (or nearly so.) I've attempted to cite appropriately, but may not have done so in all cases.

Some of the sources which were used heavily include:

- Arnie Berger's class at Washington.
  http://www.bothell.washington.edu/faculty/aberger/CSS427SP02/index.htm

Purpose

Traditionally EECS 373 students have complained about a lack of a textbook. While better books are starting to come onto the market, we feel that there isn't a book out there that is good enough to justify requiring you to buy it. This document cannot take the place of a textbook. Writing a book is an enormous task. Rather, the purpose of this document is to give you a single source of information for the ***classroom*** portion of the course.

Text

This course-pack is formated like a book in many ways. There is an index, a table-of-contents and questions at the end of some of the chapters for example. Also some non-standard features were added. For example, in many of the chapters there are highlighted questions we want you to think though before moving onwards. Because it is more educational for you to think the question though rather than just answering it, we've put the answers (or at least *a* right answer) at the end of the course-pack. These questions look like this:

[A]Why did the chicken cross the road?

The answer to the question can be found at the end of this course-pack under "A".

II. Course Introduction

EECS 373 is a class designed to teach you about embedded microprocessor systems. For a lot of you that doesn't clear up the issue too much. So let's start off by answering some fairly basic questions.

What is an Embedded System? [ABW]

- Any device, or collection of devices, that contain one or more dedicated computers, microprocessors, or microcontrollers
    - Device(s) may be local - Printer, automobile, etc.
    - Devices may be distributed - Aircraft, ship, Internet appliance
    - A PC or workstation may contain one or more embedded systems.
- Embedded computing devices have rigidly defined operational bounds. Not general purpose computers (PC, Unix workstation)

Who uses them?

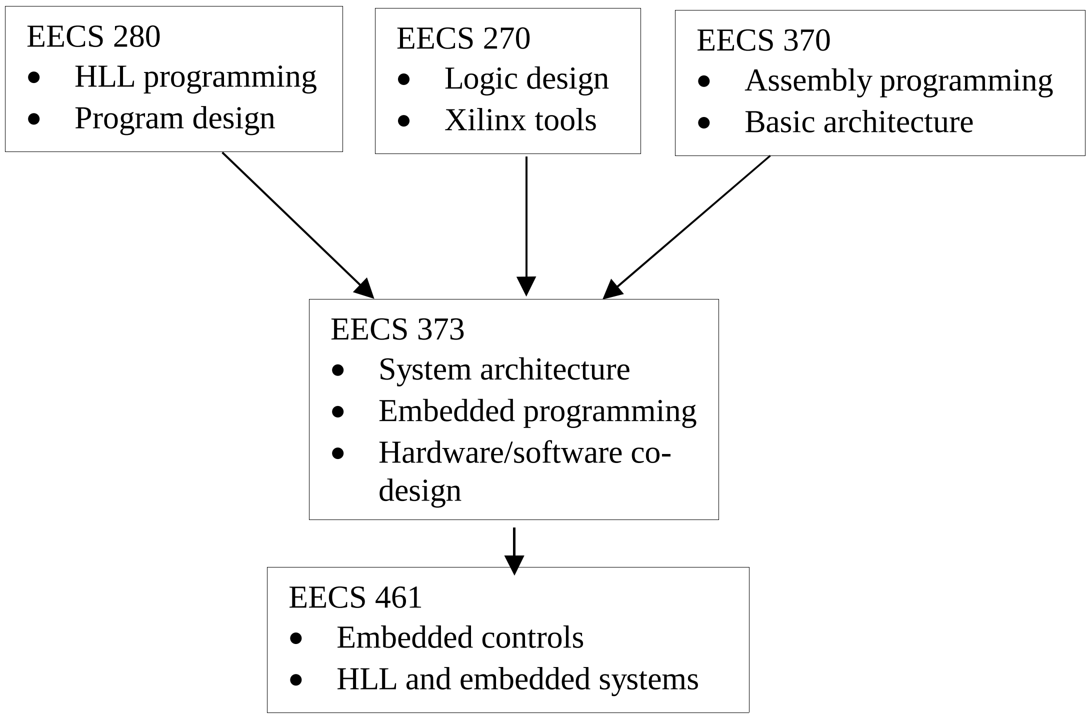
- The computer industry: Graphic cards, printers, network cards, scanners and routers.
- The home: Home thermostats, microwaves, ovens and even refrigerators are likely to have one or more embedded processors.
- In Michigan, the automotive industry is very active in the field of embedded systems. 2003 model cars can be expected to have 20-80 microprocessors controlling nearly everything (air/gas mixture in the engine, anti-lock breaks, air-bags, the CD player, and even the simple clock on the dashboard.)
- Net effect: embedded systems are everywhere.

Why study embedded systems?

- Since embedded systems are everywhere. That means there are a lot of jobs in the field.
- As processing power increases, we'll be able to do incredible things that we haven't begun to imagine.
- Because it sits on the hardware/software boarder you can write code and also play with hardware. Even if you think you enjoy one of those more than the other, it is nice to have that change of pace!
- Basically, because it's fun and rewarding.

Embedded systems at the Univ. of Michigan

There are a number of classes closely related to EECS 373 at the University of Michigan.



At the start of this class you are assumed to be familiar with material from the required classes (270 and 370 are the official prerequists, but 280 is required for 370). In particular, logic design and assembly programming will be very important. Many architectual issues from 370 will also be used as will bits of C programming.

Other Introductory issues

In lab we will be loaning you a copy of the data book (usually called "the white book") and making available a PowerPC assembly book ("the green book"). While this course pack, as well as the lectures, will discuss the MPC823 those books will be the primary source for learning about the chip and how to use it. In fact you will find the white book invaluable during lab and probably even during exams.

See the syllabus and/or web page for issues on grades, cheating, lab times, office hours, etc. This course pack doesn't cover that stuff...

III.PowerPC Assembly and Architecture

An Instruction Set Architecture (ISA) defines an interface between software and hardware. It specifies processor aspects visible to programmer such as the number and size of registers as well as the precise semantics and encoding of instructions.

An important point is that the ISA is not tied to a particular implementation (microarchitecture). For example, in EECS 370 you worked with an ISA called LC2K (or something similar). It had 8 registers and a certain legal range of memory. The format and effect of each of the machine instructions were well defined. But you wrote simulators for a number of different *implementations*. Those implementations included a single-cycle processor, a multi-cycle processor, and a pipelined processor.

The PowerPC ISA was jointly defined by IBM, Apple, and Motorola in 1991. It is used by Apple on Power Macintosh systems and is based on IBM Power ISA used in RS/6000 workstations. The chip we use in lab, the MPC823 implements a 32-bit version but has no floating point support. It is a RISC (Reduced Instruction Set Computer) ISA with some very non-RISC features. The RISC- features include 32-bit fixed-length instructions, 32 32-bit general purpose registers, and only load and stores access memory and external devices. All other instructions operate on registers. The Non-RISC features include several special-purpose registers (link for example) and a few strange instructions (my favorite is **eieio** as in the "Old MacDonald" song).

The PowerPC assembly language is *big-endian*.

Databook notation

In this chapter we'll provide some of the highlights of PowerPC assembly. The data book (white book) and the green book both provide a lot more information. The notation we use to describe the behavior of instructions is borrowed from the white book. An incomplete summary follows:

- (r4) means the contents of register 4.
- (x)0 indicates x 0's in a row.
- 0x5555 means 5555_{16} just as it does in C/C++.
- The symbol || means concatenation. So 15||0x0000 would be the same as 0xF0000.
- MEM(x,y) is a reference to y bytes at memory location x.
- r4[x-y] is a reference to bits x though y of register 4.

So using this notation, MEM(100,2) ← (8)0 || r4[0-7] would indicate that 0 zeros are concatenated with bits 0-7 of register 4. That halfword is then loaded into memory location 100. So if register 4 held the value 0x12345678, memory location 100-101 would now hold 0x0012. The complete specification is found on page B-3 of the white book.

PowerPC Assembly Example

The best way to learn an assembly language is usually to dive right in. So let's look at a simple PowerPC program. which converts an infinite uppercase ASCII string into

```
Loop:     lbz  r4,  0 (r3)
          addi r4,  r4,  0x20
          stb  r4,  0 (r3)
          addi r3,  r3,  1
          b    loop
```

Figure 1. Simple PowerPC program

lbz **r4, 0 (r3)**
loads a **b**yte and **z**ero-extends it to 32 bits effective address is (r3) + 0
data book notation: *r4 ← (24)0 || MEM((r3) + 0 , 1)*

addi **r4, r4, 0x20**
add an **i**mmediate value
r4 ← (r4) + 0x20
(0x20 is the decimal number 32 in hex. In ASCII the value for the upper and lower case of a given letter are exactly 32 apart. For example, 'a' – 'A' = 32.)

stb **r4, 0 (r3)**
stores a **b**yte
effective address is (r3) + 0
MEM((r3) + 0 , 1) ← r4[24-31]

The second **addi** has the same format as above, while **b** is an unconditional branch.

So the code in figure 1 simply loads a byte in from memory, adds 32 to the value and stores the newly incremented byte back out to the same location in memory. It then increments the register which points into memory by 1 and repeats the process forever. Obviously this is not most useful program in the world, but it does serve as a nice starting point for learning the assembly language.

PowerPC instructions

The PowerPC assembly language is fairly complex. In this section we introduce the assembly language more formally. This section does not take the place of the white and green books. Rather it is a simple overview of the language.

Loads and Stores

Load/store opcodes start with **l** and **st**, respectively. The next character indicates the memory access size: **b**yte, **h**alfword, (2 bytes), or **w**ord (4 bytes). So **lb** is a load byte instruction. The effective address for the memory access can be computed in one of two ways. The most common is "register indirect with immediate index" (also known as "base + offset" or "base + displacement"). The format used in the assembly language for this addressing mode is "d(ra)" which denotes an effective address of (ra) + d where d is a 16-bit signed value.

Another addressing mode in the PowerPC ISA is "register indirect with index", also known as "inde**x**ed" or "register + register". If you want to use this addressing mode you need to append an **x** to the opcode. So **stbx** rS, rA, rB denotes that the effective address is (rA) + (rB). One interesting note, the PowerPC architecture *sometimes* treats r0 as a constant 0 but not always. One case of where it does is in the indexed version of the load and store if rA=r0 then r0 is treated as a 0 no matter what actual value is stored in r0.

^{B}Why do you think r0 is treated the way it is for indexed loads?

One other obvious issue exists for loads: what happens when you load 16 bits into a 32-bit register? Obviously the lower 16 bits of the register will be loaded with the 16 bits from memory, but that still leaves the other 16 bits. The PowerPC gives two options you can do an **a**rithmetic load or a **z**eroing load. In the case of the arithmetic load the upper 16 bits get the value of the most significant bit of the 16 bits that are loaded. This is a 2's complement sign extension. The zeroing load always puts zeros in those upper 16 bits. You might think that something similar would exist for loading a byte, but for whatever reason, only the zeroing option is allowed by byte loads (but see the **extsb** instruction).

Finally, the PowerPC supports an update feature for its loads and stores. A load with **u**pdate has the side effect of changing the register used in computing the effective address to the value of the effective address. For example if r3 is equal to 12 the instruction

lwzu r4, 12(r3) would load register four with the value stored in memory location 24 and would change register three to have the value 24.

The green and white books have many more details about loads and stores and about support for still different formats and options. In closing we list those load and store opcodes which were discussed or hinted at in this section.

Summary of load/store instructions seen so far:

```
lbz     lhz     lha     lwz     stb     sth     stw     lbzx
lhzx    lhax    lwzx    stbx    sthx    stwx    lbzu    lhzu
lhau    lwzu    stbu    sthu    stwu    lbzux   lhzux   lhaux
lwzux   stbux   sthux   stwux
```

ALU instructions – not exactly orthogonal

When an ISA is described, one word that is often used to describe a "pretty" ISA is that the ISA is *orthogonal*. This means that each opcode has all of the options of other, similar opcodes. So if a store instruction supports byte-addressing, there should be a load instruction which does the same. With respect to ALU instructions, the PowerPC does not achieve this goal. Each operation seems to have its own quirks and particularities. In some cases there is no avoiding this. A 32-bit multiply *does* generate a 64-bit number, where a 32-bit and instruction does not. But there are cases that seem centered on reducing the number of encodings needed to implement the ISA.

Most arithmetic and logical instructions have two versions in the PowerPC ISA::

- Register-register -- **add**   r1, r2, r3 (meaning r1 ← (r2) + (r3))
- **I**mmediate -- **addi** r1, r3, 5   (meaning r1 ← (r3) + 5)

Immediate operands are limited to 16 bits but are are always expanded to 32 bits for processing. Arithmetic operations (+, -, *, /) sign extend the immediate while logical operations (and, or, etc) zero extend the immediate.

A few instructions (add, and, or, xor) have a third version:

- **I**mmediate **s**hifted -- **addis** rB, rA, 5   means   rB ← (rA) + (5 || 0x0000)

Thus the instructions **andis**, **oris**, **xoris** enable twiddling of bits in upper half of a register. The primary use of **addis** is to load a value outside the 16-bit immediate range.

Another interesting feature of the ISA is that for the instructions **addi** and **addis** (only!) r0 is treated as a zero if it is in the Ra spot. This allows for a way to load 32-bit immediates into a register using only two instructions. The simplified mnemonic for **addi** and **addis** with rA=r0=0 is **li** (**l**oad **i**mmediate) and **lis** (**l**oad **i**mmediate **s**hifted).

[C]Consider the following two ways for loading a full 32-bit value into a register:

```
lis   r3, 5                     lis    r3, 5
ori   r3, r3, 99                addi   r3, r3, 99
```

When are these two approaches not equivalent?

Subtraction is also a bit interesting in the PowerPC assembly language. The true subtract instruction is **subf**, which is **sub**tract **f**rom. In the instruction **subf** rD, rA, rB the value computed is Rb-Ra. Thus Ra is subtracted from Rb. The simplified mnemonic **sub** uses the **subf** instruction but swaps the order of rA and rB. So **subf** r4, r5, r6 is the same as **sub** r4, r6, r5. Of course there are also immediate versions of these instructions, although they are actually simplified mnemonics (**subi** for example). See page F-2 of the green book for details on simplified mnemonics for subtract instructions.

Multiplication has its own quirks. The problem is that when you multiply two 32-bit numbers you can get a 64-bit value. Since all of the registers are 32-bits in size, this presents something of a difficulty. Some ISAs have a special register where the upper 32-bits of the result are placed. Others cause the 64-bit result to be split across two successive general purpose registers. The PowerPC ISA instead has two separate instructions, one which computes the high word of the result, the other computes the low word. The basic instructions are **mul**tiply **h**igh **w**ord (**mulhw**) and **mul**tiply **l**ow **w**ord (**mullw**). Proper use of the overflow bit in the XER register (described below) can help to determine if the **mulhw** instruction is needed.

Compared to the rest of the basic arithmetic instructions, division is very straightforward. The two basic instructions are simply **divw** and **divwu**, for signed and unsigned division of words.

Other instructions

In addition to load/store and ALU instructions there are many other instructions. Obviously this includes branches. But it also includes comparison instructions and instructions that manipulate special purpose registers. For the most part, we will introduce these special purpose instructions as needed, but without the branching instructions we are hard pressed to write useful assembly programs, so we will discuss them here.

The basic instructions are the unconditional <u>**b**</u>ranch **b**, and the <u>**b**</u>ranch <u>**c**</u>onditional, **bc**. If you look in the green or white books (pages 8-23 and B-20 respectively) you will see that the unconditional branch is fairly complex. The branch offset is a 24-bit field. It can be treated as either a PC-relative address or an absolute address depending on a the value of a bit in the instruction encoding (the AA bit). Further, it can can save it's current PC to a special purpose register (the Link Register, or LR). Having the old PC around is obviously useful when branching to a function.

The **bc** instruction is even more complex. When writing assembly code, the simplified mnemonics like **blt** and **bne** are used more often than the **bc** instruction itself.

^{D}Using the white or green book, figure out what the 32-bit encoding would be for the instruction **bne cr3, bob** in the following code:

```
      bne cr3, bob
      nop
      nop
bob:  nop
```

Use relative addressing. **bne** is defined on page B-22 of the white book . (Warning, this is quite hard and requires a bit of digging.)

PowerPC Architectural issues

Trying to summarize the PowerPC architecture in a few paragraphs is doomed to failure. The green book spends hundreds of pages trying to do just that. Rather we will only look at a few of the more important architectural issues at this time. Later in this course-pack various other architecture issues will be discussed as needed.

Special Purpose Registers

If you did the **bne cr3, bob** problem above, you've probably realized that there are a number of special registers in the PowerPC ISA. In fact, the MPC823 *implementation* has a huge number of such registers. The number specified by the PowerPC ISA is only large. If you look on page Index-11 of the green book under “registers” you will see this listing of registers. Of all these registers, we will only focus on a couple of the more basic.

The CR register, as you likely learned above, is the basis for deciding if a branch should be taken or not. In the LC computer seen in EECS 370 the branch format was something like

```
BEQ 1 3 joe
```

Meaning that if registers 1 and 3 were equal in value the branch would be taken to the label **joe**. In the PowerPC the same thing is accomplished by using a **c**o**mp**are instruction (**cmp**) and a **bc** instruction. The **cmp** instruction sets 4 bit of the 32-bit CR register. Normally speaking on the MPC823 you will only use the first 4 bits, called CR0. (See page 2-5 of the green book.) Those 4 bits are

- Negative (LT) – this bit is set when the result is negative.
- Positive (GT) – this bit is set when the result is positive (and not zero).
- Zero (EQ) – This be is set when the result is zero
- Summary overflow (SO) This indicates at an overflow occurred sometime since this be was explicitly cleared. See page 2-11 of the green book for details.

So to do the same thing as the LC instruction seen above you could do:

```
cmpw r1, r3
beq joe
```

This may seem like something of a waste, but keep in mind instructions like **add** may also affect the CR register (CR0 only though) and thus the compare is often unneeded.

Registers other than the CR also can be important. The XER, LR and CTR registers can be important. You might want to read pages 2-1 to 2-11 of the green book, skiping over the details about the FPSCR (which is for floating point numbers, something the MPC823 does not support.) You will find the LR very important later on, and the CTR, used correctly, can make certain “for” loops much easier to write in assembly.

Questions

1. Which of the following would be associated with the ISA and which with a specific implementation of the ISA?
   a) Size of the L1 cache
   b) Number of general purpose registers?
   c) Number of ALUs
   d) Maximum number of bytes of memory allowed (both!)
   e) Number of bits used for the immediate value in a load instruction

2. In the base + displacement addressing mode for loads and stores, why is the displacement only 16 bits?

3. What value would be stored into what memory address given the following instructions? Assume r0=4, r1=8, r2=16, and r3=257
   a) stb r1, 0(r2)
   b) stb r0, 4(r3)
   c) stb r3, 8(r20)
   d) stw r3, 8(r20)
   e) stwx r2, r3, r1
   f) stwx r0, r1, r2

4. Indicate the value of any and all registers that change after the following instructions are executed? Assume that at the start of each instruction, r0=4, r1=8 and that memory address 0=0xFF while the other address are 1=1, 2=2, etc. (remember these are byte addresses!) Give your answers as 8-digit hexadecimal numbers.
   a) lbz r3, 0(r1)
   b) lha r3, -4(r0)
   c) lhz r3, -8(r1)
   d) lwu r3, 4(r1)
   e) lhzu r3, -4(r0)

5. Write PowerPC assembly code which performs the following instructions. Unless otherwise noted, assume that all intermediate values will fit in a 32-bit integer.
   a) R5=R1+R2+R3.
   b) R5=(R1-R2) * R3
   c) R6=(R1+16)/12
   d) R3=(R3-R4)-R5 " you may not use any simplified mnemonics.
   e) R6=(R1*R2) / R4. "The final result will fit into R6 but the intermediate result may take more than 32-bits.

6. Write a PowerPC assembly program that loads 100 words starting at memory location 0x03330000 and sums them in order (lowest memory address to highest). If any of the intermediate additions cause overflow, R1 is set to a 1. Otherwise it is set to the value of the sum of the 100 words. You may not use the CTR register.

7. As #6 above, but you must use the CTR register for the main loop.

8. Write a program which compares the values in R1 and R3 and sets CR7 and CR0 to the result of this comparison.

IV.Memory Mapped I/O – the basics

How does one access I/O devices though a programming language? You've learned at least one assembly language and I'll bet there wasn't a “talk to disk drive” instruction. Instead of developing special instructions to talk to I/O devices, the vast majority (all?) microprocessor use memory mapped I/O. The idea is that load and store assembly instructions (or other instructions which can talk to memory on CISC machines) can be used to read and write messages to the devices.

Each device has one or more special purpose registers that are each mapped to some address in memory. For example, imagine a mouse that needs to be able to report where it has moved since that last time it was queried. Imagine that the mouse responded with a 16-bit field, and 8-bit two's complement number describing the amount moved in along the X-axis, and the other 8-bits would be the two's complement value of the amount moved along the Y-axis. So 0x0070 would imply the mouse was moved up along the Y-axis but not at all along the X-axis. We could design the device so that it supplied this information whenever memory address 0x0000E004 was loaded. That is, the data for that load would not come from the DRAM but rather from the mouse.

Continuing the mouse example, say a half-word load is performed from 0x0000E004 and the response was 0x0400. This might imply that the mouse has moved some positive distance along the X-axis since it was last queried. If another half-word load is performed immediately afterward, it may be that a 0x0000 would be returned as the mouse likely would not have moved enough to be noticeable in the few tens of nanoseconds between the two loads.

This example, while fairly simple, sheds light on a number of important issues regarding memory-mapped I/O devices. First of all, consider what would happen if the data from these requests were cached. Each time a read was performed the data would be the same as the last time the device were read rather than the data that was supposed to be read from the device. Obviously this is undesired behavior, and so we have to be sure that those addresses mapped to I/O devices are marked as being *uncacheable*.

Secondly, notice that performing a read can have *side-effects*. In our mouse example, the I/O device responds with the movement since the last time it was read from. This means that in addition to responding with the requested data the future behavior of reads from the device has been impacted. This is important because we now have to be careful about issuing loads speculatively (for example if a branch has been predicted taken, we now have to be sure that a load directed at an I/O device doesn't issue until the prediction has been verified, otherwise we might lose some data. By the same token, we have to worry about issuing loads of the proper size. If we load a word when we only need a half-word, it is possible that the extra bytes loaded are from an I/O device. Again, we might lose data from our mouse because of the extra read.

Lastly, and perhaps most obviously, memory is now *volatile*. Not volatile in the since that the data goes away without power (although that might be true also), but rather volatile in the since of the C-language keyword. Obviously this is closely related to being uncacheable.

Volatile (C-language definition): the variable may be affected/changed by outside influences.

Simple memory-mapped I/O example

Say we have two simple devices. One is a button, the other a light-emitting diode (LED). Associated with the button is an 8-bit register located at memory address 0xF0040004. That 8-bit register returns a 0x01 if the button is pushed, and 0x00 if the button is not currently pressed. Similarly, associated with the LED is an 8-bit register located at memory location 0x000000FF. If a zero is written to that memory location the light is turned off. Writing anything else to the register causes the light to come on (and stay on until turned off). Write a PowerPC assembly program that turns the light on when the button is pressed and turns it off when the button is not pressed. It should keep checking forever.

```
          lis r3, 0xF004
          ori r3, 0x0004
          li  r4, 0x00FF
loop:     lbz r5, 0(r3)
          sb  r5, 0(r4)
          b   loop
```

[E]Now write the code so that the button acts as a toggle. Each time you press the button the light should change state. Start the light in the off state.

Questions

1. Assume you have the following I/O devices
   - A button which is memory mapped into a single byte of memory at location 0x0000F000. That byte will be a 1 if the button is pressed, a zero otherwise.
   - A keyboard that is memory mapped into a halfword of memory at location 0xFFFF0000. That halfword will be all ones if nothing has been pressed since the last time a read was performed. Otherwise it will be the 8-bit ASCII value for the character pressed (the 8 most significant bits will be 0's in that case.)
   - An array of 8 LEDs that are memory mapped into a single byte at memory location 0x0000F001. Each LED is mapped to 1 bit of that byte. If the corresponding bit for an LED is 1, it will be turned on. Otherwise it will be off.

   a) Write a program that will turn on all of the LEDs if the last key pressed was an X. (0x58).

   b) Write a program which lights all of the LEDs only if the key currently pressed is different than the key which was last pressed.

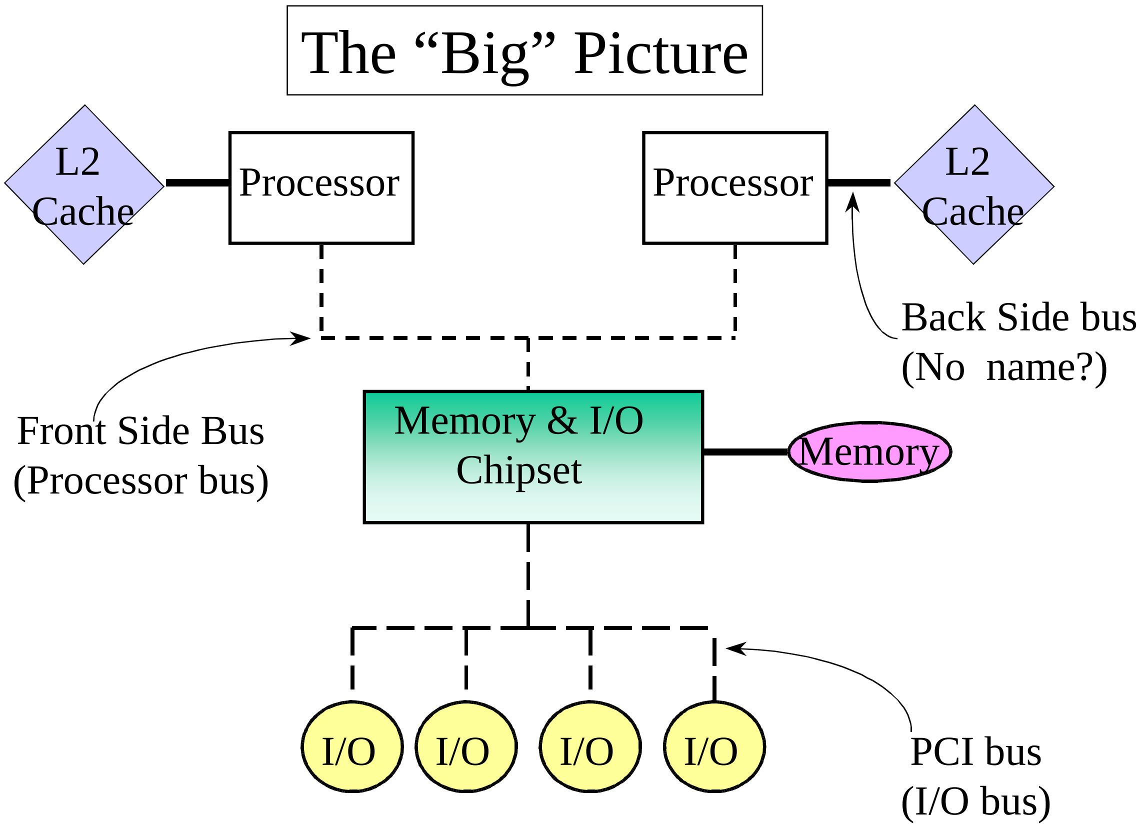
   c) Write a program which displays the most recently pressed character (in ASCII) on the LEDs only if the button is pressed. Otherwise all of the LEDs should be off.

2. In your own words, list some of the advantages of memory-mapped I/O over using special purpose instructions to communicate with the I/O devices. List one or more disadvantages.

V. Bus Protocols

When interconnecting different devices some type of protocol needs to be observed. Traditionally the protocol used in an embedded system is that of the processor's memory bus. I/O devices are connected directly to the same interconnect that the memory system uses. Looking back the discussion about memory-mapped I/O in Chapter 4, having the I/O devices watching the memory bus makes sense. After all, the I/O devices are acting as if they were memory devices – you can read and write to them using loads and stores, just as is done with memory. So the I/O device can intercept loads and stores to those addresses to which they are mapped.

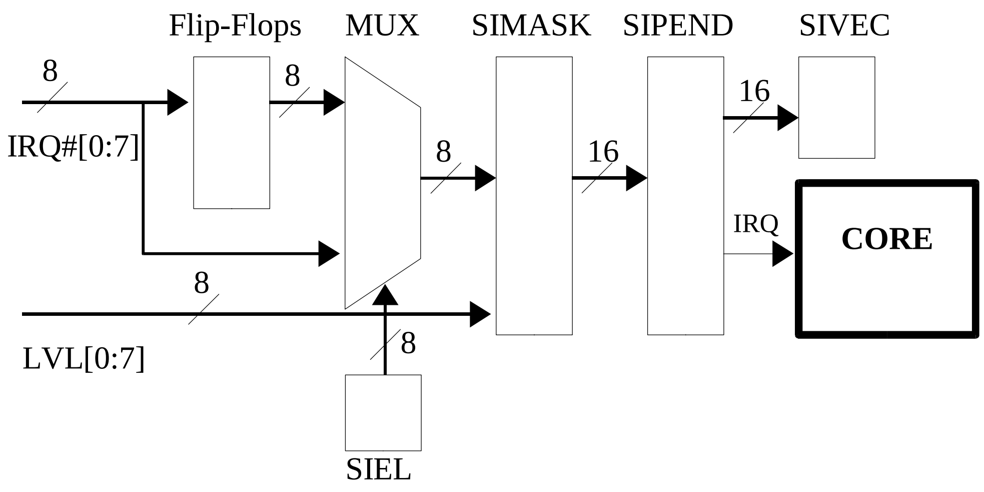
Over time, there have been many variations on this basic theme. One of the more interesting (and relevant) examples of such variation can be seen in desktop PC arena. When the 486



Memory mapped I/O – again

VI.ABI and the Stack

VII.Interrupts



VIII.DMA – Direct Memory Access

IX.Counters and Timers

X.Analog and Digital Signal

XI.Bonus Topics

Memory types

Error correction

Parity

Error Correction Codes (ECC)

Viterbi algorithm for error correction

The basic idea:

- With each data bit, send a parity bit along also.
- The parity bit is the parity of *all* of the data sent so far.
- Assume that there are few bit errors. If the parity doesn't work out, figure out the **smallest** number of bit flips which could have occurred to get to the current data.

Example:
Say we want to send the data 10001.
Using even parity we might actually send:
Notice that *all* of the data bits thus far, plus the *current* parity results in an even number of ones. So at the first step we have 1 for data and 1 for parity. At the second step we have 10 for data and 1 for parity. At the 5th and final step we have 10001 for data and 0 for parity (still even number of ones).

Say that in transmission two errors occur and we get:

Notice that both the data and the parity bit of the 3rd step have been changed.

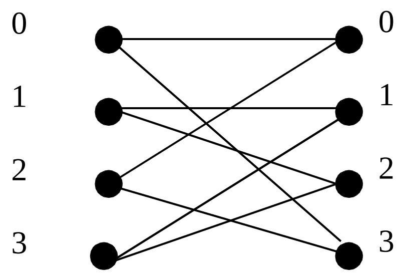
We want to *try* to recover the data. Notice that the parity bits for the 4th and 5th step are clearly wrong. Thus we know we had some error.

Topic shift

Lets consider writing the data and parity bits as a single decimal number with the data bit as the MSB. So the code that was sent was 31112 and the code received was 31212. Notice that in a legal encoding certain numbers **cannot** follow other numbers. For example a 0 (that is 0 data 0 parity) cannot be followed by a 2. If it were the parity bit of one of the two numbers must be wrong The basic reason is that if we have a 0, the parity up to that time must have been even and so we put in a zero parity bit. If the next data bit is a 1 then the parity bit that goes with that data bit must also be a 1. . If you don't believe me try to find a case where a 2 can follow a 0.

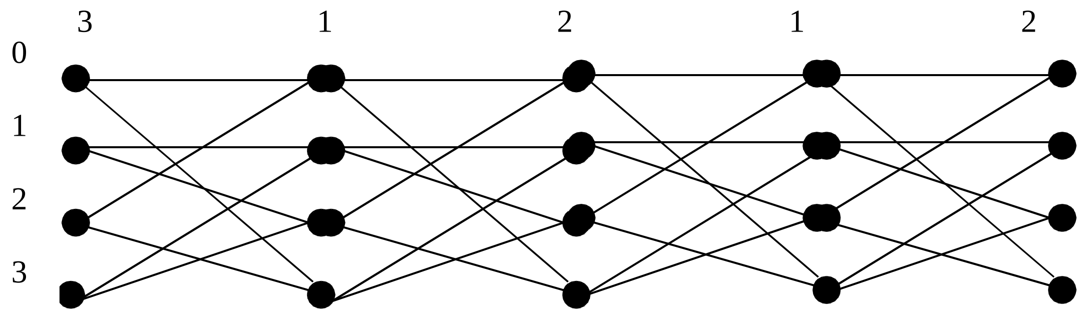
In general 0 and 2 can be followed by 0 or 3 and 1 and 3 can be followed by a 1 or a 2.

We can draw a graph which shows the same thing as follows:



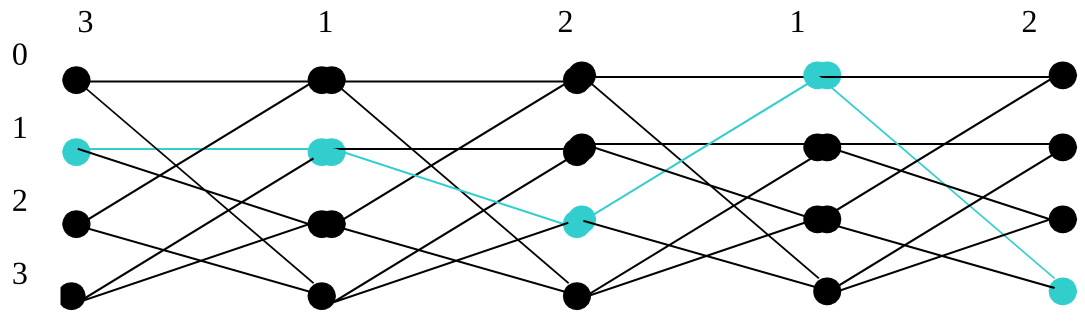
The numbers on the right indicate the starting point, the numbers on the left the finishing point. So 0 followed by a 3 is legal, but 3 followed by a 0 is not.

Now, let us return to the original problem. 31212 has a transition from 2 to1 which is clearly not allowed. So we have at least one error. Let us draw a graph for the entire problem.

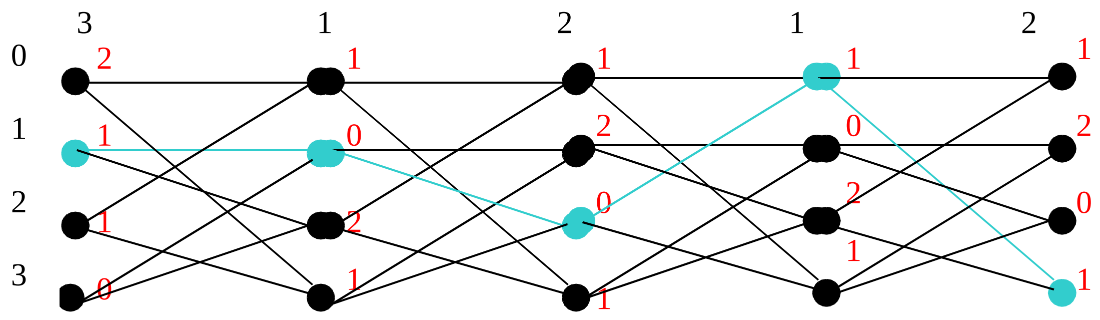


On the left the numbers indicate the value we would *guess* was sent at any given step. On the top we have the values we actually received at that step.

So if we chose the following path (in blue) we would be claiming the actual message sent was 11203.



Now we need to consider the weight associated with each path though this graph. We weight each of the nodes based upon the number of bit flips required for our data received to jive with the data we think was sent. For example, if we received a 0 (which is 00 as a bit string) but we are guessing we got a 2 (10 as a bit string) then there must have been one bit flip. So we associate each node with the number of bits that would have had to have flipped to get the value we are guessing. Again, using the same example, we get:



Where the numbers in red are the number of bit flips which would have had to occurred. So in our proposed solution (in blue) the answer number of bit flips which this solution would require is 1+0+0+1+1=3. Not bad, but we happen to know there is at least one valid solution which only has 2 bit flips. After all, we created our input that way.

So what we need to do is find the shortest path which traverses this graph. In general there are somewhere around 2^N different paths where N is the number of data bits (or steps) sent. Searching all possible paths is possible for small values of N, but very difficult for large values of N. (at N=100 the universe would end before you searched all of the possible paths one at a time even using the greatest computer in the world.)

As such we take advantage of the principle of optimality and notice that if a given node is in the shortest path, the shortest path to that node from the left, and from that node on the right, must also be in the shortest path. Put another way, if we just start to traverse the graph and, at each "step" keep track of *a* shortest path to get there, we can solve the problem in no time.

For example, at step 1 we can quickly figure out that starting at node 3 is of weight 0, node 2 or 1 is of weight 1 and node 0 is of weight 2. During step 2 we consider the two ways we could get to node 0. We must be coming from either 0 or 2 from step 1. The weights associated with those two nodes is 2 and 1 respectively. So we know the minimal way to get to node 0 in step 2 is from node 2 in step 1. The total weight is 2 (1 for node 2 of step 1 and 1 for node 0 of step 2.)

So from the above we get:

So our answer is 31232. Which sadly is **not** what was originally sent. But it is the best guess. Remember, this algorithm doesn't promise much of anything. But it *usually* does a good job.

	Step1		Step2		Step 3		Step 4		Step 5	
Received	3		1		2		1		2	
	path	cost	path	cost	path	cost	path	cost	path	cost
Node0	0	2	20	2	320	2	3120	1	31200	2
Node1	1	1	31	0	311	2	3111	2	31111	4
Node2	2	1	32	1	312	0	3112	4	31232	1
Node3	3	0	23	2	323	3	3123	1	31202	2

Data	1	0	1	0	1
Parity	1	1	0	1	0

Data	1	0	0	0	1
Parity	1	1	1	1	0

Software engineering in embedded systems

Alphabetical Index

Embedded System 4
ISA 6

A To get to the other side!

B I don't have a clue. Certainly this makes it easier to use a register as a pointer, but you'd think that using the register indirect form with an offset of zero would allow this also.

C If the immediate being added has a 1 in the most significant bit or if the number loaded into the register with the **lis** instruction has a 1 in the MSB, then the add instruction will treat the numbers as negative. This will cause the **addi** to have a different result than the **ori**.

D This is the equivalent of **bne 4, 14, bob**. The 4 means "branch if the condition is false." This is the 001zy found in the table on page B-21 of the white book. The 14 indicates the 14th bit of the Condition Register (CR). It is the Zero (also called EQ) bit of CR3. This can be best seen on pages 2-5 and 2.6 of the green book, although hints of it can be found on page 6-22 of the white book. So we get:

16	BO=4	BI=14	BD=3	AA=0	LK=0
0 5	6 10	11 15	16 29	30	31

This corresponds to 0b010000 00100 01110 00000000000011 0 0 = 0x408E000C
The value of BD=3 is because the next instruction executed is BD instructions in front of the current PC. NIA and CIA are defined in table 8-3 in the green book.

E Your answer may be different. I've not tested this so let me know if you think it doesn't work!

```
        lis r3, 0xF004
        ori r3, 0x0004
        li  r4, 0x00FF
        li  r5, 1         ; set r5.  Next state for light is on.
        li  r6, 0         ; r6 is the last known button state.  Assume
                            unpressed at start
        sb  r6, 0(r4)     ; turn off light!

loop:   lbz r7, 0(r3)     ; read button
        cmp r7, r6        ; ? Has button changed?
        beq loop          ; if not, then nothing todo!
        mr  r7,r6         ; this is an or instruction (page F-23, green book)
                          ; notice it does not change the CR.
        blt change        ; If r7<r6 button went from pressed to unpressed.
                          ; light does not change.
        sb  r5, 0(r4)     ; change light state
        nor r5, r5, r5    ; \  bitwise_negate(A)+2 = !A if A starts
        addi r0, r5, 2    ; /  as either 1 or 0.  Anyone have a better way?
        b   loop
```