# Contents

## 1 Introduction

1.1 Cpr E 210 Review

1.1.1 Karnaugh Maps (K-Maps)

1.1.1.1 Example

1.1.2 FSM – Finite State Machine

1.1.2.1 Example

1.1.3 TTL Review

1.1.3.1 TTL Datasheet

1.1.3.2 Sourcing vs. sinking

1.2 Introduction to Microcontrollers

1.2.1 Embedded Programming

1.2.1.1 Real-time software

1.2.2 Binary Review

1.2.2.1 Example

## 2 C Programming

2.1 Variable Names & Types

2.1.1 Arrays

2.1.1.1 Example

2.1.1.2 Multi-dimensional Arrays

2.1.1.3 Initializing Arrays

2.1.2 Strings

2.2 Variable manipulation

2.2.1 Bitwise Operators

2.2.1.1 Example

2.2.1.2 Example

2.2.1.3 Group Exercise

2.3 Boolean Flow Control

2.3.1 if, else if, else

2.3.1.1 Example

2.3.2 Comparison (Relational) – Numeric

2.4 Expressions & Bitwise Operations

2.4.1 Bit-testing

2.4.1.1 Example

2.4.1.2 Example

2.4.1.3 Group Exercise

2.5 Functions

2.5.1 Return Types

2.5.2 Parameters

2.5.3 Prototyping

2.5.3.1 Example

2.5.4 Calling a Function

2.5.4.1 Example

2.5.4.2 Passing Variables

2.5.5 Global vs. Local

2.6 Looping

2.6.1 Bit-masking
# Overview of Computer Architecture

## 3.1 Types of Buses

### 3.1.1 Connecting to a bus

### 3.2 CPU Registers

## 2.12 Debugging Techniques

### 2.12.1 Printf

### 2.12.2 IDE - Integrated Development Environment

### 2.12.3 Other Tools

### 2.12.4 Problems with Debugging

## 2.11 Structs

### 2.11.1 Designing a struct

### 2.11.2 Group Exercise

## 2.9 Pointers

### 2.9.1 Using pointers

### 2.9.2 More on Pointers

### 2.9.3 Embedded Programming Example

### 2.9.4 Pointer Math

### 2.9.5 Array Notation

### 2.9.6 Dynamic Memory Allocation

### 2.9.7 Example – Dynamic Memory Allocation

### 2.9.8 Memory Dump Example

### 2.9.9 Group Exercise

## 2.8 Compiler Statements

### 2.8.1 #include statement

### 2.8.2 #define statements

### 2.8.3 #ifdef statements

### 2.8.4 Example 1

### 2.8.5 Example 2

### 2.8.6 Group Exercise

## 2.7 General Input/Output Statements

### 2.7.1 printf

### 2.7.2 sprintf

### 2.7.3 scanf

## 2.6 Typecasting

### 2.6.1 Array access

### 2.6.2 For Loop

### 2.6.3 While Loop

### 2.6.4 Loop Control

### 2.6.5 Switch Statement

### 2.6.6 Example

### 2.6.7 Group Exercise

### 2.6.8 Example 2 – Dynamic Memory Allocation

### 2.6.9 Memory Dump Example

### 2.6.10 Use of sizeof

## 2.5 Structures

### 2.5.1 Example

### 2.5.2 Group Exercise

## 2.4 Debugging Techniques

### 2.4.1 Sanity Checks

### 2.4.2 IDE - Integrated Development Environment

### 2.4.3 Other Tools

### 2.4.4 Problems with Debugging

## 2.3 Typecasting

### 2.3.1 Example

### 2.3.2 Group Exercise

## 2.2 Compiler Statements

### 2.2.1 #include statement

### 2.2.2 #define statements

### 2.2.3 #ifdef statements

### 2.2.4 Example 1

### 2.2.5 Example 2

### 2.2.6 Group Exercise

## 2.1 General Input/Output Statements

### 2.1.1 printf

### 2.1.2 sprintf

### 2.1.3 scanf

## 1 Overview of Computer Architecture

### 1.1 Types of Buses

### 1.2 Connecting to a bus

### 1.3 CPU Registers
4 PowerPC Assembly Language

4.1 Assembly Instructions ................................................................. 56
4.2 Assembly Terminology ................................................................. 57
4.3 Addressing Modes .................................................................... 57
  4.3.1 Immediate Addressing .......................................................... 57
    4.3.1.1 Example .............................................................................. 58
  4.3.2 Indexed Addressing ............................................................... 58
    4.3.2.1 Example .............................................................................. 58
  4.3.3 Effective Address for a Load or Store Instruction ................... 58
    4.3.3.2 Assembly Language .......................................................... 52
    4.3.3.1 Machine Language ............................................................ 52

4.4 Labels ....................................................................................... 64
  4.4.1 Labels for Functions ............................................................. 64

4.5 Basic Assembly Instructions ....................................................... 65
  4.5.1 Data Movement ................................................................. 65
    4.5.1.1 Immediate Load ............................................................... 66
    4.5.1.2 Memory Load ................................................................. 67
    4.5.1.3 Memory Load – Arrays .................................................. 69
    4.5.1.4 Memory Store ............................................................... 70
    4.5.1.4 Math Operations ........................................................... 75
  4.5.2 Data Manipulation ................................................................. 72
    4.5.2.1 Logical Operations ......................................................... 72
    4.5.2.2 Group Exercise .............................................................. 73
    4.5.2.3 Example ........................................................................... 73
    4.5.2.4 Math Operations ........................................................... 75
5 I/O Subsystems ................................................................................................................... 106

5.1 A/D I/O Subsystem ......................................................................................................... 107
  5.1.1 Converting Analog to Digital ..................................................................................... 107
  5.1.2 Terminology and Equations ...................................................................................... 108
  5.1.3 ADC Implementation ............................................................................................... 108
    5.1.3.1 Successive Approximation ............................................................................ 108
  5.1.4 ADC on the PowerPC .............................................................................................. 109
    5.1.4.1 Lab Setup ...................................................................................................... 110
    5.1.4.2 Using the QADC64 .................................................................................... 110
    5.1.4.3 Queued ADC ............................................................................................ 111
    5.1.4.4 Programming the ADC ............................................................................... 113
    5.1.4.5 Example ....................................................................................................... 114
  5.1.5 ADC on the PowerPC .............................................................................................. 109
  5.1.6 Link Register ............................................................................................................ 110
  5.1.7 Timer ........................................................................................................................ 110
  5.1.8 Using the QADC64 .............................................................................................. 111
  5.1.9 Examples .................................................................................................................. 111

5.2 Interrupts ......................................................................................................................... 114
  5.2.1 Terminology ............................................................................................................ 114
  5.2.2 Interrupt vs. Polling ............................................................................................... 115
  5.2.3 More on Interrupts .................................................................................................. 116
  5.2.4 Different Types of Interrupts ................................................................................... 116
  5.2.5 Vectors .................................................................................................................... 117
  5.2.6 More on Exceptions .............................................................................................. 117

5.3 Assembly Flow Control ................................................................................................. 79
  5.3.1 Branching ............................................................................................................... 80
  5.3.2 Example .................................................................................................................. 80
  5.3.3 Comparison ............................................................................................................. 81
  5.3.4 Longer Example ..................................................................................................... 83
  5.3.5 Yet Another Example ............................................................................................ 84
  5.3.6 Advanced Flow Control ......................................................................................... 86
  5.3.7 Link Register .......................................................................................................... 88
5.2.7 Interrupt Handling........................................................................................................ 117
5.2.8 Interrupts and the Stack ............................................................................................ 118
5.2.9 Interrupt Activation....................................................................................................... 118
  5.2.9.1 I/O Subsystem Setup ............................................................................................ 119
  5.2.9.2 I/O Subsystem Operation .................................................................................... 119
5.3 Periodic Interrupt Timer (PIT)...................................................................................... 119
  5.3.1 Configuring the PIT for Interrupt Operation .............................................................. 122
  5.3.1.1 Example .............................................................................................................. 123
5.3.2 Writing the ISR ........................................................................................................... 124
  5.3.2.1 Lab Code .............................................................................................................. 124
5.4 MPC 555 Interrupt System ............................................................................................ 126
  5.4.1 Initialization Steps ..................................................................................................... 126
    5.4.1.1 Step 1: Module Specific Initialization ................................................................. 126
    5.4.1.2 Step 2: Level Assignment ................................................................................... 127
    5.4.1.3 Step 3: Enable Interrupt .................................................................................... 127
    5.4.1.4 Step 4: Set Appropriate Mask Bits in SIMASK .................................................. 127
    5.4.1.5 Final Step: Setting MSR[EE] and MSR[RI] Bits ................................................ 127
  5.4.2 Initialization Code....................................................................................................... 128
1 Introduction

1.1 Cpr E 210 Review

Basic logic – AND, OR, NOT, XOR, NAND, NOR

Combinatorial logic – logic combines to give a result
Logic can be simplified by either algebraic reduction or other techniques

1.1.1 Karnaugh Maps (K-Maps)

Technique to simplify logic
Map output onto grid based on input
Left, right, up, down, change by only one input
One K-Map per output equation desired
Produce Sum of Products (SOP) by circling ones

To produce a SOP:
  Circle each 1 at least once
  For each circle that circles multiple 1’s, a term is dropped
  Must be circled in powers of 2 (1, 2, 4, 8)
  Cannot circle diagonally
1.1.1.1 Example

3 input (A,B,C) yields a single output (X)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

K-Map

Circle all 1’s at least once

A=0 Row
C changes between 0 and 1
Drop C and take the other two inputs
1st term = !A B

B=1, C=1 column
A changes between 0 and 1
Drop A and take the other two inputs
2nd term = BC
A=1, B=0, C=0 term
Only one to circle, no simplification
Take term literally
3rd term = A!B!C

Result
\[ X = !AB + BC + A!B!C \]

Required logic
- 2 2 input AND gates
- 1 3 input AND gate
- 1 3 input OR gate

1.1.2 FSM – Finite State Machine

- Used to create logic circuits with memory (i.e. they remember what happened previously)
- Uses components called flip-flops
- Flip-flop holds a value and only changes on a given clock signal (rising edge, falling edge)

\[ Q \] is the value of the flip-flop
\[ Q^+ = \text{Next value of } Q \]

Typically most flip-flops have a Q and \(!Q\) output

Flip-flop examples: D, JK

**D Flip-Flop**
\[ Q^+ (\text{Next State}) = D \] where \(D\) is the input to the flip-flop

**J-K Flip Flop**
- \(J\) is equivalent to the S (Set), when \(J=1, K=0, Q^+ \text{ always} = 1\)
- \(K\) is equivalent to the R (Reset) when \(J=0, K=1, Q^+ \text{ always} = 0\)
- When \(J=1, K=1, \text{equivalent to a toggle}, Q^+ = !Q\)
- When \(J=0, K=0, Q \text{ stays the same}\)
1.1.2.1 Example

Design an FSM to output a 1 when the sequence of 1 0 is seen across the input, hover in the final state

Draw the state diagram

![State Diagram]

3 states -> $2^N$ states can be captured by $N$ flip-flops

3 states therefore we need 2 flip-flops, $2^N = 4$ which is greater than the 3 required states

Draw a truth table

3 inputs – 2 from the flip-flops (current state) ($Q_0$, $Q_1$)

1 input (1 0 sequence) (I)

Next state is a combination of the current state and input via combinatorial logic

<table>
<thead>
<tr>
<th>$Q_0$</th>
<th>$Q_1$</th>
<th>I</th>
<th>$Q_0+$</th>
<th>$Q_1+$</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

$X$ is a don’t care, i.e. it does not matter if it is a 0 or a 1

Why do we have X’s in this truth table?

Notice that we do not use State 11, so we don’t care what happens in that situation
State 0 (S0)  See a 0 (I=0)
State diagram tells us to stay in state 00
Q0+ = 0, Q1+ = 0, X = 0

State 0 (S0)  See a 1 (I=1)
State diagram transitions to state 01
Q0+ = 0, Q1+ = 1, X = 0

3 outputs – 3 K-Maps

K-Map for Q0+

\[
\begin{array}{c|cccc}
| \ \ & 00 & 01 & 11 & 10 \\
\hline
0 & 0 & 1 & \times & 1 \\
1 & 0 & 0 & \times & 1 \\
\end{array}
\]

Q0+ = Q1 + Q0!Q1

K-Map for Q1+

\[
\begin{array}{c|cccc}
| \ \ & 00 & 01 & 11 & 10 \\
\hline
0 & 0 & 0 & \times & 0 \\
1 & 1 & 1 & \times & 0 \\
\end{array}
\]

Q1+ = !Q0I
Draw logic diagram for FSM with flip-flops and logic

### 1.1.3 TTL Review

- Transistor-to-Transistor Logic
- Operates at +5 V (Digital 1), 0 V (Digital 0)
- Further details in EE 333
- Based on the bipolar transistor
- First developed by Texas Instruments in 1965
- Simple logic, AND, OR to counters to buffers

**SN74LS69**

- SN Series Number 74
- L Low Power
- S Schottky
- Part number 69

#### 1.1.3.1 TTL Datasheet

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>V&lt;sub&gt;cc&lt;/sub&gt;</td>
<td>Source voltage</td>
</tr>
<tr>
<td>Gnd</td>
<td>Ground for transistors in chip</td>
</tr>
</tbody>
</table>

#### 1.1.3.2 Sourcing vs. sinking

- Sourcing – Current is provided through the chip via the V<sub>cc</sub> of the chip
- Sinking – Current is sunk by the chip by driving its output to ground

TTL is better at sinking rather than sourcing

### 1.2 Introduction to Microcontrollers

Recall the parts of a computer: CPU, memory, I/O
• Microprocessor - A single chip that contains the CPU or most of the computer
• Microcontroller - A single chip used to control other devices

Examples:
- Microprocessor - Pentium, PowerPC chip in your computer
- Microcontroller - 68HC11, 68332, MPC555

A microcontroller is essentially a microprocessor with several other features embedded onto a single chip

Examples of things that use microcontrollers
- Automobiles, Automatic Cameras, CD player, etc.

Why use a microcontroller?
- Reduce chip count
- Many applications do not require as much computing power
- Reduced power consumption
- Reduced design cost

In fact, industry sells 10 times as many microcontrollers as microprocessors

What are the parts of a microcontroller?
- CPU
- Memory
- I/O (Input/Output)

• CPU
  Central Processing Unit
  “Smart part” of the computer that processes data and makes decisions
  Has all the parts of a normal microprocessor

• Memory
  RAM – Random Access Memory – Storing data while microcontroller is running
  ROM – Read Only Memory – Store bootup data information
  EEPROM or EPROM – Persistent storage of data parameters that can be rewritten

  Example: Alarm clock saving the time when the power goes off

• I/O
  Methods to interact with the world outside the microcontroller

  A typical CPU takes up only a small portion of the actual silicon real estate of a microcontroller leaving additional space for other features.

Examples:
A/D – Analog to Digital Converter
Temperature Sensor
Display controller
Timing circuits
Communication circuits
Parallel, Serial, Ethernet

1.2.1 Embedded Programming

Key Points in Embedded Programming
- Code Speed: Timing constraints, limited proc. power
- Code Size: Limited memory, power, physical space

Programming Methods
- Machine Code
  - Low level language: Assembly
  - High level language: C, C++, Java
  - Application level language: Visual Basic, scripts, Access

Why use C in embedded programming?
- Fairly efficient
- Provides an additional level above assembly programming
- Supports access to I/O
- Ease of management of large embedded projects

Why use assembly?
- High speed, low code size
- However, difficult to do a large project in assembly

1.2.1.1 Real-time software

- Software that must read input changes when they occur and change outputs predictably, within a certain time limit
- Real-time software programming is typically driven by interrupts
- Not necessarily fast code – simply has to meet time constraints
- Three classes of real-time systems
  - Hard real-time = Failure results in a catastrophe, loss of human life
    - Nuclear reactor, airplane
  - Soft real-time = Failure results in loss of data, loss of value
    - Airline reservation system, stock market
  - Best-effort = No penalty for missing time
    - Email, web surfing

1.2.2 Binary Review

- Smallest unit is a bit
Base 2 notation -> Two values
1 (TRUE) 0 (FALSE)

- Nibble
  4 bits – 16 possible values
  Ratio of 1 Nibble to 1 Hexadecimal character

- Byte, Word, Double Word
  8 bits, 16 bits, 32 bits

Three most common forms of notation
- Decimal (base 10) 0,1,2,...,9
- Hexadecimal (base 16) 0,1,2,...,9, A,B,C,D,E,F
- Binary (base 2) 0,1

Another form is octal (base 8)

Converting between forms
- Binary to Hexadecimal
  Easy, each 4 bits is a hexadecimal character
  11000100
  
  1100 0100
  0 x C 8

Key Point: Remember, a notation is just a way of representing a specific quantity. A number is not in hex or in decimal or in binary form. Hex, decimal, and binary are just ways of representing a specific quantity. It is up to you to decide how to deal with the variable and how the information is represented.

1.2.2.1 Example

Passed in the value 50 in an 8 bit quantity

Binary = 00110010
Hex = 0x32
Decimal = 50

Could be the actual number 50
x = x + 50

Could be various bits of information
If bit 6 is set, do this

Could be a combination
If bit 6 is set, x = x + lower nibble of the value
2 C Programming

Course Prereqs: Com Sci 207/227
Get the recommended book on C as a brush-up

ANSI C – Standard for C compilers across the world

2.1 Variable Names & Types

- Can have long variable names
  - X in FORTRAN vs. Area, Graph2, InFile, etc.
- No punctuation marks besides underscore
- Must start with a letter
- Case Sensitive
  - MyVariable is not the same as myvariable
- Use long variable names
  - H vs. nHeight
  - Fi vs. InputFile
  - A vs. fArea
- Can you remember what a variable was used for 1 year from now, 6 months from now?
- For looping variables, use common looping names
  - j, k
- Spend a little more time now = savings later on when debugging
- Use a naming convention to help quickly identify variables (cover later)

<table>
<thead>
<tr>
<th>Name</th>
<th>Bytes</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char</td>
<td>1</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1</td>
<td>0 to 255</td>
</tr>
<tr>
<td>Short</td>
<td>2</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>Int</td>
<td>varies</td>
<td>may be same as short</td>
</tr>
<tr>
<td>Long</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Float</td>
<td>4</td>
<td>7 significant digits</td>
</tr>
<tr>
<td>Double</td>
<td>8</td>
<td>15 significant digits</td>
</tr>
<tr>
<td>* (pointer)</td>
<td>width of memory</td>
<td>Range of memory</td>
</tr>
</tbody>
</table>

- Floating point = IEEE 754 standard
- used infrequently
- very expensive versus integer operations
- Ex: 68HC11 floating point op -> 160 cycles vs. integer 3 cycles
- char is one of the most used types in embedded programming
- Single byte of memory
• Don’t think of it as an actual character, think of it as the 8 bits required to represent a character

2.1.1 Arrays

Sequence of a specific variable type stored in memory
Zero-indexed (starts at zero rather than one)
Define an array as

\[
\text{Type } \text{VariableName}[\text{ArraySize}];
\]

• Last element is found at \(N-1\) location

\[
\text{int } \text{nMyIntArray}[30];
\]

\[
\text{nMyIntArray}[0] \quad /* \text{The first element of the array}*/
\]

\[
\text{..}
\]

\[
\text{nMyIntArray}[29] \quad /* \text{The last element of the array}*/
\]

\[
\text{nMyIntArray}[30] \quad /* \text{INVALID! Beyond the edge of the array}*/
\]

• Be careful of boundaries in C
  No guard to prevent you from accessing beyond array edge
  Write beyond array = Disaster

• What exactly is an array?
  Not a specific type
  Pointer to a block of memory
  No built-in mechanism for copying arrays

2.1.1.1 Example

\[
\text{int } \text{nTestArray1}[20]; \quad /* \text{An array of 20 integers}*/
\]

\[
\text{int } \text{nTestArray2}[20]; \quad /* \text{An array of 20 integers}*/
\]

\[
\text{nTestArray1}[0] = \text{nTestArray2}[0]; \quad /* \text{This works}*/
\]

\[
\text{nTestArray1} = \text{nTestArray2}; \quad /* \text{This does not work}*/
\]

2.1.1.2 Multi-dimensional Arrays

Declared the same as normal arrays with an extra set of brackets
• Think of it as \([\text{row}][\text{col}]\)
• Example:
  \[
  \text{char DblArray}[20][50];
  \]
Declares 20 sets of 50 byte arrays
i.e., a 20 row x 50 column array

\[
\text{DblArray}[5][0]
\]

DblArray[5] will create a pointer to the 6th row in the
array whereas no brackets will create a pointer to the entire
block of memory

### 2.1.1.3 Initializing Arrays
- Can initialize an array just like a normal variable
- Example:
  
  String
  
  ```
  char szTemp[] = "Some string";
  ```

  Values
  
  ```
  int nTemp[] = {5,15,20,25};
  ```

  Letters
  
  ```
  char szTemp[] = {'A','B','C','D'};
  ```

  Double Dimensioned
  
  ```
  char szTemp[2][] = { {'A','B','C','D','E'},
                      {'U','V','X','Y','Z'} }; 
  ```

### 2.1.2 Strings
- What is a string?
  Special array of type `char` that is ended by the NULL (\0) character
- Remember to create an array of N+1 characters to allow space for the NULL
  character
  ```
  20 character string
  char szString[21]; /* 20 + 1 */
  ```

- Why is there a NULL character?
  Otherwise, how can you tell what is good and bad in a string?

### More on Variables
- What is a variable?
  Each variable is just a block of memory
  Block of memory that equates to a certain value
  Actual value is determined by the programmer
  Integer, Byte, A few bits, etc.
- Example:
  The ASCII character ‘A’
  Actually the numeric value 65
  In hex = 0x41
  Depending on the debugger, it may appear as
  ‘A’, 65, or 0x41
- Array Example
2.2 Variable manipulation

- Standard operators
  +, -, /, *
- Specialized operators
  \%
  Mod
- Space-saving operators
  Combine = with an operator
  \nVal += 10; /* Adds 10 to nVal */

Increment/decrement operators
  ++ or --
  Pre-increment and post-increment

  \nVal++; /* Adds 1 to nVal */
  \nVal = nTemp++ + 1; /* Adds after expression is evaluated */
  \nVal = --nTemp * 6; /* Sub before expression is evaluated */

  \nNote: Parens have no effect
  \n(nVal++)-4; is the same as \nVal++ - 4;

2.2.1 Bitwise Operators

Refer back to the Cpr E 210 review.

Bit manipulation is a key component of embedded programming.

- Why?
  \nSpace – Instead of using 8 bits to store one value, now we can use individual bits to store information
- Operations are done on a bit by bit basis
  Hence the name bitwise operators/manipulation
- Hex notation is the most common form used for bit manipulation
  0xFF is 11111111 in binary
  0x10 is 00010000 in binary
  Recall the binary -> hex conversion from earlier
- AND Operator & – Clear bits and Test bits
  0 ANDed with anything will always give a zero
  1 ANDed with anything will give the same value
To clear bits:
1. Set the bits you want to clear to zero  (Clear)
2. Set all other bits to 1  (Preserve)

2.2.1.1 Example
Clear bits 2,3 of an 8 bit number

\[
\begin{array}{c|c|c|c|c|c|c|c}
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & F & 3
\end{array}
\]

\[\text{byVal} = \text{byVal} & 0xF3;\]

- OR Operator  \(\mid\) (pipe symbol)– Set bits
  1 ORed with anything will always give a one

\[
\begin{array}{c|c|c|c|c|c|c|c}
0x10 & 0x10 & = & 0x10 \\
0x01 & 0x10 & = & 0x11 \\
0xFF & 0x00 & = & 0xFF
\end{array}
\]

To set bits:
1. Set the bits you want to make a 1 to 1  (Set)
2. Set all other bits to zero  (Preserve)

2.2.1.2 Example
Set bits 7,5 of an 8 bit number

\[
\begin{array}{c|c|c|c|c|c|c|c}
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0xA0
\end{array}
\]

\[\text{byVal} = \text{byVal} & 0xA0;\]

- Exclusive OR operator  \(^\) - Toggle bits
  1 XORed with anything will toggle the bit
  Not the power sign, to do powers, use the math library

\[
\begin{array}{c|c|c|c|c|c|c|c}
0x10 \ ^\ ^0x10 & = & 0x00 \\
0x01 \ ^\ ^0x10 & = & 0x10 \\
0xFF \ ^\ ^0x00 & = & 0xFF
\end{array}
\]

To toggle bits:
1. Set the bits you want to toggle to 1 (Toggle)
2. Set all other bits to zero  (Preserve)
• Inversion operator ~
  ~ (0x10) = 0xEF;

• Shift operators
  Used to shift a sequence of bits to the left or right
  >> or <<

  Syntax
  
<table>
<thead>
<tr>
<th>Variable/Value</th>
<th>Operator</th>
<th># of Places</th>
</tr>
</thead>
<tbody>
<tr>
<td>nVal = nVal &gt;&gt; 4; /* Shift nVal to the right 4 places */</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  Why use the shift operator?
  Count the number of ones in a bit
  Iterate through each bit in a loop

  Note: The shift operation may be done via an arithmetic shift or by a logical shift
  Arithmetic – MSb stays the same on a right shift
  Logical – Always shift in a zero

  0x0F >> 2 = 0x03;
  0x0F << 2 = 0x3C;

2.2.1.3 Group Exercise

Suppose we have the following definition:
  short nVal;

Write the code to do the following:
  Set bit 13
  Clear bit 4
  Toggle 15, 14

2.3 Boolean Flow Control

Flow Control – Making the program behave in a particular manner depending on the input given to the program

Why do we need flow control?
  Not all program parts are executed all of the time
  i.e. we want the program to intelligently choose what to do

Statements for Boolean flow control
  if, else if, else

Key Point:
The evaluation for Boolean flow control is done on a TRUE / FALSE basis. TRUE / FALSE in the context of a computer is defined as non-zero (TRUE) or zero (FALSE).

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-1, 5, 15, 225, 325.33</td>
<td>TRUE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>FALSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.3.1 if, else if, else

- Must always have if
- May/may not have else if or else
- Syntax

```c
if ( Condition1)
{
  ... 
}
else if (Condition2)
{
  ... 
}
else if (Condition3)
{
  ... 
}
else
{
  ... 
}
```

- Follows a level hierarchy
  - `else if` statements are only evaluated if all previous `if` and `else if` conditions have failed for the block
  - `else` statements are only executed if all previous conditions have failed
- Why is how if statements are evaluated important?
  - Helps in the design of efficient logic
  - Know if a condition is evaluated, all previous conditions up to that point have failed
    - For example, in the above syntax example, the `else if (Condition2)` will only be executed if `Condition1` is false.

2.3.1.1 Example

```c
if ( nVal > 10) 
{
  nVal += 5;
}
else if(nVal > 5) /* If we reach this point, */
{    /* nVal must be <= than 10 */
  nVal -= 3;
}
else /* If we reach this point, */
{    /* nVal must be <= than 10 and */
  nVal = 0; /* nVal must be <= than 5 */
}
2.3.2 Comparison (Relational) – Numeric

Standard operators

>  
<  
>=  
<=  
== Equality
!= Not Equals

• Gives a result of zero (FALSE) or non-zero (TRUE)

Key Point:
A TRUE result may not necessarily be a 1

• Equality Double equals sign ==
  = Assigns a value
  == Tests for equality, returns non-zero or zero

Consider:
if (nVal == 5) versus if (nVal = 5)

The second expression always evaluates to TRUE
Why?

Comparison – Multiple Conditions

• Tie together using Boolean operators
  && AND
  || OR
  ! NOT

• Examples:
  if ( (nVal > 0) && (nArea < 10))
  if( (nVal < 3) || (nVal > 50))
  if ( ! (nVal <= 10) )

• Conditions are evaluated using lazy evaluation
  Lazy evaluation – Once a condition is found that completes the condition, stop
  Ex. OR any condition is found to be TRUE 1 OR anything = 1
  AND any condition is found to be FALSE 0 AND anything = 0
• Why is lazy evaluation important?
  Makes code run faster – skips unnecessary code
  Know condition will/will not evaluate, why evaluate other terms
  Can use lazy evaluation to guard against unwanted conditions
  Checking for a NULL pointer before using the pointer

2.4 Expressions & Bitwise Operations

Remember, conditions are evaluated on the basis of zero and non-zero

The quantity 0x80 is non-zero and therefore TRUE

\[
\text{if (3 || 6)}
\]

is a valid expression, not very useful but valid

2.4.1 Bit-testing

• If we store information in bits, we need to be able to test for bits being a 1 or 0
• Recall use of bit-wise AND to clear bits and test bits
• Why does this work?
  0 AND anything = 0          Clears bits
  1 AND value = same value    Tests to see if bits are set, i.e., TRUE
  If the result is non-zero, that means that at least one of the bits was set since any value
  ANDed with a 1 is the same value
• How do we test?
  Set the bits we wish to test to a 1
  All other bits are set to zero
• Test for a single bit
  1. Set the bit to a 1 that you wish to test for
  2. Do a bitwise AND with the value to be tested

2.4.1.1 Example

Find out if bit 7 is set
Bit 7 = 0x80 in hex

\[
\text{if ( nVal & 0x80 )}
\]

{  
  ...
}

• What happens when we want to test for multiple bits?
  Remember, an if statement looks only for a non-zero value
  Bit-wise AND does ops on a bit by bit basis
  Therefore, a non-zero value means at least one bit is set to TRUE

2.4.1.2 Example

See if bits 2 or 3 are true
Bits 2,3 = 0x0C in hex

```c
if ( nVal & 0x0C)
{
    Some code...
}
```

Now, let’s take a look at what happens for several values of nVal

<table>
<thead>
<tr>
<th>nVal</th>
<th>bit status</th>
<th>Result</th>
<th>Truth</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x04</td>
<td>2 is set</td>
<td>0x04</td>
<td>TRUE</td>
</tr>
<tr>
<td>0x0A</td>
<td>3,1 are set</td>
<td>0x08</td>
<td>TRUE</td>
</tr>
<tr>
<td>0x0C</td>
<td>2,3 are set</td>
<td>0x0C</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

Why does this present a problem?
What happens if we want to see if both bits 2 and 3 are set, not just to see if one of the bits is set to true?
It won’t work without some other type of test

Two solutions
Test each bit individually
```c
if ( nVal & 0x08 && nVal & 0x04)
```
Check the result of the bit-wise AND
```c
if ((nVal & 0x0C) == 0x0C)
```

Why do these solutions work?
Individual – Easy, individual test
Result – The result will only equal 0x0C if bits 2 and 3 are set

2.4.1.3 Group Exercise
Consider the following definition:
```c
short      nValue;
```

Write the if statements to test for the following:
- Bit 13 is true
- Bit 7, 4, or 0 is true
- Bits 15 and 0 are true
- Bits 4 and 2 are false

2.5 Functions

Goal – Calculate some value or do some task

Subroutines – May/may not return a value

Syntax

```c
ReturnType FunctionName(Type Parameter1Name, Type Parameter2Name, …)
{
```

CprE 211 Lecture Notes - 24 - 2001-2002
return (expression of ReturnType);
}

main function is the startup point for all C programs
main()
{
}

2.5.1 Return Types
• void No Return Value
• May return any variable type but an array
• Note: Don’t return a pointer to a local variable (more later)
• Examples
  i.e. return (0);
  return (nVal);
  return; /* void function */
• return keyword immediately exits the function

2.5.2 Parameters
• May have zero or more parameters
  Typically, standard practice is to keep the number of parameters below 5 to 8
• Any type, even an array
  void PassArray (char szString[])
• For an array, may or may not declare size
• If the size is not declared, make sure to either know the size ahead of time or to pass the size in as a parameter
• All parameters are local variables, i.e. altering the local variable does not affect the actual caller unless the variable is a pointer
  Arrays are passed in as pointers

2.5.3 Prototyping
• How does C look up a function?
  C \rightarrow Top down compilation
  Only knows about what it has seen so far
  i.e at line 20, knows contents of lines 1-20
  Problem: Write the function definition at the bottom, call it at the top
  Solution 1: Move the function definition earlier
  Solution 2: Write a prototype
• Prototype – Tells the compiler the function is defined somewhere in the code
  If the function is prototyped but not defined, linker error
• Prototype
  Declaration or header line of function – up to first curly brace
  Take a copy of the declaration line and add a semicolon
  No semicolon = compiler expects function body (i.e., code)
  Semicolon = prototype
2.5.3.1 Example

```c
void WritePrototype (char szString[], short nStringLen)
{
}
```

The prototype for the function is:

```c
void WritePrototype (char szString[], short nStringLen);
```

2.5.4 Calling a Function

- Syntax
  ```c
  FunctionName (parameter1, parameter2);
  ```

2.5.4.1 Example

```c
if(x > 5)
  WritePrototype(szName,20);
```

2.5.4.2 Passing Variables

Can pass via one of two ways

- Pass to be read only (Write – No effect)
- Pass allowing changes (Write – Changes actual variables)

- Pass by value (“call by value”), i.e. no changes
  ```c
  void DoValue (int, float, char);
  DoValue (5, 2.5, ‘A’);
  DoValue (nTest, fPressure, byInput);
  ```

  Value – A local variable on the stack

- Pass by pointer (“call by reference”), i.e., allow changes
  ```c
  void DoChanges (int *, float *, char[]);
  DoChanges (5, 2.5,"test"); /* Can’t do this, need a variable to use */
  DoChanges(&nTest,&fPressure,szName);
  ```

  In order to allow changes to the variable, must pass as a pointer

  Memory Address – Access to actual variable itself

- Why do we want to do this?
  Return more than one variable
Allow changes, allow function to manipulate variable
  i.e. initialize a structure, clear an array

• Why does this happen?
  Local variable
    Created on the stack
    Visible only to the function
    Enter the function: Space is created
    Exit the function: Space is destroyed
      Not really destroyed, just changed to garbage status
  Why was returning a pointer to a local variable bad?
    Return a value – OK – actual value and mechanisms are set up for that
    Return an address – Address to memory that may/may not be garbage

2.5.5 Global vs. Local

• Global variable
  Declared outside of all functions
  May be initialized upon program startup
  Visible and usable everywhere from .c file

• What happens when local/global have the same name?
  Local takes precedence

• Summary
  Local – declared inside of a function, visible only to function
  Global – declared outside all functions, visible to all functions

• What happens when you want a local variable to stick around but do not want to use a global variable?
  Create a static variable
  Syntax:
    static Type Name;
  Static variables are initialized once
  Think of static variables as a “local” global
    Sticks around (has persistence) but only the function can access it

2.6 Looping

Bit-masking Example
Write a test to check a 16 bit value (short) to see if bits 12 and 2 are off

2.6.1 Bit-masking
Why do we want to do a bitmask?
  Suppose we want to get at a value inside of a variable
    i.e. bits 1,2,3 contain a number from 0 to 7
  Already know the basic concepts
Set bits to read to 1
Set bits to ignore to 0

2.6.1.1 Example

Add 5 to an upper nibble (char byVal)
First – get the upper nibble
Set upper bits to 1, lower to 0
1 1 1 1 0 0 0 0 = 0xF0
Next – do a bitwise AND
byVal & 0xF0
Will this next expression work?
(byVal & 0xF0) + 5
No
5 is in the lower nibble but the value we just bit-masked is in the upper nibble
To solve this:
Move the upper nibble down to the lower nibble via the shift operator
((byVal & 0xF0) >> 4) + 5

2.6.1.2 Group Exercise

byStatus                   8 bits
nLevel                   16 bits
nResult                  16 bits

When bit 5 of byStatus is true, add upper 8 bits of nLevel to nResult and return the result
Otherwise, return 0

Prototype of function is int SetLevel (char, int, int &);

2.6.1.3 Example

if ( (byStatus & 0x10) == 1) will always fail

If byStatus is equal to 0x10
(0x10 & 0x10) == 1
0x10 == 0x01     False

2.6.2 For Loop

for (j=0; j<10; j++)
for (j=9; j>=0; j--)
for (fVal=0.5; fVal<10.5; fVal += 0.5)
Warning:
for (j=0; j<10; j--)
Will cause an infinite loop

2.6.2.1 Array access
Sum an Array

float fSum;
fSum = 0;

for (j=0; j<10; j++)
{
    fSum += fArray[j];
}

If you use a variable as the boundary check, be careful not to change the variable
for (j=0; j<nElems; j++)
{
    nElems = ........ /\ Can cause problems */
}

2.6.3 While Loop

while (Condition)
{

while (x < 10)
{

do
{
    }
while ( Condition );

2.6.4 Loop Control
break statement
Exits to end of loop
Essentially a goto statement that breaks out of the loop
Valid with while, do/while, for
Not valid with if

continue
Repeats loop to beginning of loop

for (j=0; j<10; j++)
{ for(k=0; k<10; k++)
{
    if (j == k)
        continue; /* Goes back to k loop check */
}
}

2.6.5 Switch Statement

switch (nVal)
{
    case 0:
        break;
    case 1:
    case 2:
        break;
    default:
        break;
}

Cannot do a switch on string or range
Produces the same code as a large sequence of if, else if statements

2.7 General Input/Output Statements
Three basic functions
printf, sprintf, scanf

2.7.1 printf
Most basic form of output
String formatted

Include files

#include <stdio.h>       /* for printf - normal C*/
#include <f1board.h> /* for lab */

Syntax
printf("X is equal to %d", x);

Outputs
X is equal to 5

Types
%c     Character     ASCII representation of number
%d Integer
%x Hexadecimal
%f Float
%e Exponential
%s String Must be null terminated

Special Chars
\n Return (Linefeed)
\r Carriage Return
\t Tab
%% % sign
%c With ASCII value of double quote to print out a double quote

Formatting
%5d 5 character wide number
%20s 20 characters wide

Decimal
%5.2f Overall Width. Decimal Places
Ex.
12.345 → 12.35
2.30 → 2.30
356.23 → 356.2

Special
%#x Adds 0x
%02d Left padding with zeros, i.e. 00:00:02
%ld Long integer

2.7.2 sprintf
Prints output to a string
sprintf(szTemp, “X is equal to %d”, x);

2.7.3 scanf
Read input into variable
Must use address of variable
& -> Gives address of variable
&nVal /* Memory address of nVal */

Same inputs as printf
scanf(“%d”, &nVal); /* Reads an integer */

2.8 Compiler Statements

Denoted by #
Used to specify special commands to compiler only
Do not result in actual binary code, instructions on how to compile
2.8.1  **#include statement**

```c
#include <……. h>
```

Directly inserts a file  
Equivalent to copy/paste

**Note**: *Do not include a .c file*

Sample include headers  
stdio.h  
stdlib.h  
string.h  
math.h

2.8.2  **#define statements**

```
#define    NAME         Value (Optional)
#define    MAX_SIZE     256
```

Style  
All Caps  
Located at beginning of file or in header file

Why use #define?  
Avoid magic numbers 20 vs. MAX_TANK_LEVEL  
Easy to change, 20 changes vs. 1 change

When compiling, compiler replaces #define with actual value

2.8.3  **#ifdef statements**

```
#ifdef    Variable
#include Some include file
#endif
```

2.8.3.1  **Example 1**

```
#ifndef __HEADER_H
#define __HEADER_H

/* Some header file code */

#endif
```

2.8.3.2  **Example 2**

```
#ifdef DEBUG
```

CprE 211 Lecture Notes - 32 - 2001-2002
/* Debugging info */
#endif

2.8.3.3 Group Exercise

Write a function that accepts an 8 bit input and prints out the binary representation of the number
i.e.        print     11010010
2.9 **Pointers**

- Points to a spot in memory
- Pointer size is dependent upon addressability of system, not type of variable that is being pointed to
- HC11 - 16 bit memory addressable
  - char * 16 bit memory address
  - long * 16 bit memory address
  - float * 16 bit memory address

- `sizeof` function
  - Returns the size in bytes of a variable
  - What would this be useful for?
    - Figuring out sizes of a variable on a system (i.e. int)
    - Calculating the size of a block of memory

**Example**

```c
sizeof (char)  = 1
sizeof(char *) = 2
sizeof(long *) = 2
sizeof(long *) = 4
```

- Pointer syntax
  - Simply the variable type with an asterik (*)
    ```c
    VariableType * VarName;
    char * pChar;
    ```
  - Contains a memory address
    - i.e. where does the pointer point
    ```c
    pChar = 0x1000;
    ```
  - * - Dereference a pointer
    - Read/write to what the pointer is pointing to
    ```c
    x = *pChar + 5;
    ```
    - Can write to a pointer's location
    ```c
    *pChar = 0x0E;
    *pChar = *pChar & 0x80;
    ```

- Use with variables
  - Need to get the memory address of the variable
  - Remember: The memory address of any variable is always the same size because it is dependent upon the system, not the size of the variable type that it is pointing to.
  - Use the & operand in front of the variable to get the variable address
    ```c
    int nVal;
    int * npVal;
    ```
npVal = &nVal; /* Address is 0x2000 */
nVal = 10;

npVal is 0x2000
*npVal is 10

*npVal = 5;
npVal is still 0x2000
*npVal is 5
nVal is 5

2.9.1 Using pointers

Three key steps when using pointers:

1. Declare the pointer
   
   type * pName;

   char * pChar;
   long * pHistory;

2. Initialize the pointer
   
   In order to use the pointer, we need to point it somewhere

   pChar = (char *) 0x1800;
   pHistory = &lValue;

   The (char *) tells the compiler this is a 16 bit memory address, not a 16 bit value.

3. Access the pointer (Read/Write)
   
   In order to get the value, we must use a * in front of the pointer

   n = *pChar & 0x80;
   if(*pHistory + 25 > TOL_HISTORY)
       *pHistory = TOL_MINIMUM;

   NULL/Bad pointers are the leading cause of system crashes

2.9.2 More on Pointers

- What exactly is a pointer?
  A memory address (16 bits, 32 bits, etc.)
  HC11 - 0x2005
  The variable itself contains a memory address
  char * pChar;

  pChar is the memory address in the variable
  *pChar accesses the memory address in the variable
Remember: Pointers do take up space and you can manipulate pointers just like normal variables.

- What does the pointer point to?
  Depends upon the system
  May not always be RAM

Two types of architecture
- Unified Memory - Motorola
  All devices, RAM, etc. share the same address space
  0x2000 may be memory, a temperature sensor, hard disk
- Split I/O - Intel
  Separate addresses for I/O and memory
  Hard disk, PCI cards - One address scheme - I/O
  Special assembly instructions to access
  Memory - Other address scheme

Note: What you may write out to the memory location may not always be what you will read back in.

Why is that?
- Device - Serial Port
  Write - Out Buffer
  Read - Reads from the In Buffer

  A device can choose to respond however it wants to reads and writes

  Thus, a write with bit 7 set may behave differently than a write with bit 7 clear

### 2.9.3 Embedded Programming Example

Given:
- Temperature 0x2500 float
- AC 0x2520 byte

If temp > 80 then turn on AC by setting bit 0 to true

```c
float * pfTemp;
char * pAC;

pfTemp = (float *) 0x2500;
pAC = (char *) 0x2520;

if ( *pfTemp > 80 )
  *pAC = *pAC | 0x01;
```
2.9.4 Pointer Math

Operates differently than normal math
Changes position by the size of the type of the pointer
   A char changes by 1 byte each time since a char is 1 byte
   A long changes by 4 bytes each time since a long is 4 bytes
Why is that?
   Want to read the next element
   But, the next element starts X bytes later

Formula to calculate new position
Start + sizeof(type) * N

where
Start = current memory location pointed to
N = Number of elements to move

2.9.4.1 Example

```c
short * pShort;
char * pChar;

pChar = (char *) 0x2000;
pShort = (short *) 0x2004;

pChar -> 0x2000
pShort -> 0x2004

pChar+1 -> 0x2000 + sizeof(char) * 1
         0x2000 + 1 * 1
         0x2001

pShort+2 -> 0x2004 + sizeof(short) * 2
           0x2004 + 2 * 2
           0x2008
```

Now, combine this with pointer notation to do a base + offset

```c
*(pChar+5) -> 0x2000 + sizeof(char) * 5
           0x2000 + 1 * 5
           *(0x2005)
           Read 1 byte (char) at memory location 0x2005

*(pShort-2) -> 0x2004 - sizeof(short) * 2
              0x2004 - 2 * 2
              *(0x2000)
              Read 2 bytes (short) at memory location 0x2000
```
*(pShort++ +5) -> Post increment so it is the same as
*(pShort + 5)
pShort++

0x2004 + sizeof(short) * 5
0x2004 + 2 * 5
*(0x200E)

New value of pShort

0x2004 + sizeof(short) * 1
0x2004 + 2 * 1
0x2006

2.9.4.2 Memory Dump Example

Consider the following memory dump

2000 FA 00 0A FC 0B FC AB CA
2008 BC DD DE AC AF F0 08 33

Examine the previous expressions (leave off the post-increment)
*pChar 0xFA
*pShort 0x0BFC
*(pChar+5) 0xFC
*(pShort-2) 0xFA00
*(pShort+5) 0x0833

2.9.5 Array Notation

Use [ ] to access an element in an array

An array is a block of memory
float fArray[20];

fArray[0] = 5;

The array name without any brackets give a memory address
float * pFloat;

pFloat = fArray;

pFloat = &fArray; /* No! Invalid, address of an address */
pFloat = & fArray[1]; /* Address of element 1 */
An array is just a pointer to an allocated block of memory
The brackets simplify and remove the need to use the *
Can use both * and [ ] interchangeably

```
pFloat = fArray;

*(fArray+3)
    is the same as
pFloat[3]
```

### 2.9.5.1 Group Exercise

Write a function named GetData that copies N items of data into a pointer

<table>
<thead>
<tr>
<th>nItems</th>
<th>16 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataRead</td>
<td>0x2040</td>
</tr>
<tr>
<td>plArray</td>
<td>32 bit memory address</td>
</tr>
</tbody>
</table>

```
void GetData (long * plArray, short nItems)
{
    long * pData;    /* Need a pointer to the memory location */
    short   j;

    /* Initialize the pointer */
    pData = (long *) 0x2040;

    for(j=0; j<nItems; j++)
    {
        *(plArray+j) = *pData;
    }
}
```

### 2.9.6 Dynamic Memory Allocation

Allocating/freeing memory as needed in a program

```
#include <malloc.h>

void * malloc (int nBytes);
void free (void *);
```

Successful malloc - Returns a non-zero memory address
Failed malloc - Returns a NULL (zero)
2.9.6.1 Use of sizeof

Allocate a block of 200 bytes (char)

char * pTemp;

pTemp = (char *) malloc (200 * sizeof (char));

sizeof allows us to make sure we allocate 200 of the actual type
sizeof – C keyword – gives us the size of the type
However, not the size of the block allocated
sizeof(pTemp) yields 2
because a char * is 2 bytes (16 bits)

Once memory is allocated, you are responsible for freeing the memory as well

free (pTemp); /* Release memory back to system */

- Advantage that you do not have to know in advance what the size will be
- Requires OS support
- Very time consuming

2.9.6.2 Example – Dynamic Memory Allocation

Read X data points from
ValvePos 0x2000 float

/* return type is float * - returning a memory address –
start of the data */

float * GetDataBlock (int nPoints)
{
    float * pValvePos; /* Location of ValvePos */
    float * pData; /* Data block to allocate */
    int j; /* Index variable */

    /* Initialize pointers */
    pValvePos = (float *) 0x2000;

    /* Allocate the block of memory */
    pData = (float *) malloc (sizeof(float) * nPoints);
    /* Read the data in from ValvePos */

    for(j=0; j<nPoints; j++)
    {
        pData[j] = *pValvePos;
    }
/* Return start address of memory */

    return pData;

} /* NOTE : It is the responsibility of the callee to free the memory */

2.10 Typecasting

Typecasting is done in the form of

    ( type )
    (float) 25
    (char *) 0x100A

Tells the compiler to treat a variable in a certain way
In short, you tell the compiler explicitly what to do
Allows for silencing of warnings
Allows you to force the compiler to do a specific behavior
Essentially, “trust me, I know what I’m doing”

Why typecast?
Convert a memory address to a different type
malloc returns a void *
cannot dereference a void *, no type associated with it
typecast it to whatever type we want since all pointers are 16 bit (HC11)

2.10.1 Example
Write a float out an 8 bit serial port

0x1040  8 bit Serial Port

Problem : Float is 5 bytes (HC11) and the serial port is only 8 bits

Solution: Use typecasting to make the memory byte accessible

    float    fVal;  /* Actual value */
    char *   pByte;  /* Byte accessibility */
    char *   pSerial; /* Serial output */
    char j;

    fVal    = 10;

    /* Initialize pointers */

    pSerial = (char *) 0x1040;
    pByte  = (char *) &fVal;
/* &fVal is a float *
typecast this memory address to change it from float
accessible to byte accessible */

for(j=0; j<sizeof(float); j++)
{
    *pSerial = *(pByte+j);
}

2.11 Structs

- Used to group related data together
- Similar to C++ classes except that there are no constructors or functions on a struct

```c
struct Rectangle
{
    int Top;
    int Bottom;
    int Left;
    int Right;
};

Rectangle TestRect;
/* may need to use struct keyword before listing a struct */
/* struct Rectangle TestRect only in declaration */

Rectangle * pRect;
pRect = & TestRect;

TestRect.Top = 10;
TestRect.Bottom = 20;

(*pRect).Left = 5;
/* can also be done as */
pRect->Right = 10;
```

2.11.1 Designing a struct

Given:
- FieldDevice

Name is a string of 20 characters width
Address is a 5 byte quantity
Status is a 1 byte quantity
struct FieldDevice
{
    char Name [21];
    char Address[5];
    char Status;
};

2.11.1.1 Group Exercise

Read the serial port at 0x1040 and return the actual data. The 1st byte read
details the number of data bytes

char * ReadSerial ( )
{
    char * pSerial;
    char * pData;
    char DataSize;

    /* Set up serial port pointer */
    pSerial = 0x1040;

    /* Read 1st byte from serial port - Number of data
    bytes to follow */
    DataSize = *pSerial;

    /* Allocate block of memory */
    pData = (char *) malloc ( DataSize * sizeof (char) );

    /* Read serial port DataSize times */
    for ( int j=0; j<DataSize; j++)
    {
        pData[j] = *pSerial;
    }

    /* Return the populated block of data */
    return pData;
}

Remember, in the Motorola domain, a memory address may not necessarily be
memory, i.e. it may be a device that has some hardware read cycle like a serial
port, a hard disk, video board, etc.
2.12 Debugging Techniques

2.12.1 Printf
- Most common debugging technique
- Display certain values
- Can also evaluate results of conditional statements

2.12.1.1 Sanity Checks
- Add in checks to make sure variables are within reasonable bounds
- Ex: j is -3205 when it is only supposed to be positive
- Often used in the case of pointers
  - Check to see if a pointer is NULL

2.12.2 IDE - Integrated Development Environment
- Contains compiler, editor, debugger all in one package
- GUI - Graphical User Interface
- Breakpoints
  - Interrupt at each code line
  - Executes to a certain point and then stops
  - Does not work with time-critical applications
- Stepping
  - Step through code one C instruction at a time
- Call Stack
  - Examine who called which function with which arguments
- Variable Inspection
  - Get a snapshot of the current values of a variable

2.12.3 Other Tools
- In-chip simulator
  - Entire chip simulated
  - See all registers, pins, values, etc.
- Remote debugging
  - Useful for driver debugging

2.12.4 Problems with Debugging
- Debugging takes time (CPU cycles)
  - Can change behavior of program
- Time-critical and interrupt-driven apps are difficult to debug

-------------------------- End of C Notes ------------------------------------------
3 Overview of Computer Architecture

- Microcontroller
  - CPU + other devices all on a single chip
  - Need some way to talk/interact with devices
    - Bus - Collection of wires with a common function

3.1 Types of Buses

Address bus
  - Who are we talking to
  - PowerPC 555
- A31..A0 -> 32 bits
  - 32 bit memory addressable (~ 4GB)
- HC11
  - A15..A0 -> 16 bits
  - 16 bit memory addressable ~ 64k
- x86 - 8086
  - 20 bits ~ 1 MB
- Unidirectional - CPU chooses who to talk to

**Data bus**
- What we are saying
- Actual flow of data between CPU and device
- Power PC 555
  - D31..D0 -> 32 bits
  - Numbered in PPC manual with
    - D0 (MSb) - Most significant bit
    - D31 (LSb) - Least significant bit
  - 32 bit processor - 32 bit datapath
- HC11
  - D7..D0 -> 8 bits
  - 8 bit processor (8 bit datapath)

**Control bus**
- How the conversation occurs
- Protocol - rules for conversation
- Enforces order when talking to devices
- R/W - Read/Write
  - Reading in from device (input)
  - Writing out to device (output)
  - Relative to CPU
- Other control pins
  - Interrupt, Chip Select

### 3.1.1 Connecting to a bus

- 3 types of buses
  - Address - Who is talking
  - Data - What is being said
  - Control - Protocol (rules) for talking
- How do multiple people talk on the same bus?
  - Problem : 1 device (1), 1 device (0) \(\rightarrow\) result = garbage
  - Need three states: 1, 0, not talking
  - Tri-state buffer
0 - Enable
1 - High Impedance

<table>
<thead>
<tr>
<th>Input (I)</th>
<th>Enable</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

X = High Impedance state, looks like an open circuit

3.2 CPU Registers

- Store everything in memory
  - Each operation - go out to memory, bring in, manipulate, write out
  - Problem: Slow
- Solution: CPU Registers (Register File)

3.2.1 Register

- Set of high-speed flip-flops
- Runs at speed of CPU, no refresh required
- Internal to CPU, visible to CPU only
  - Bring information in from memory to register
  - Manipulate inside of registers
  - Write result back when you are done
- Three basic operations in a register-based CPU
  - Memory to register
  - Register to memory
  - Register to register
  - Note: No memory to memory copy without special devices
3.2.2 **CPU Organization**

- **Register file - Internal CPU registers**
  - Number and type of registers
    - HC11 – A, B, X, Y, etc.
    - PowerPC – r0, r1, r2, r3, … r31

- **ALU - Arithmetic Logic Unit**
  - Actual ops (Add/subtract/shift)
  - Recall CprE 210 – shifter, adder
  - Two inputs to the ALU
    - **Opcode** - What operation to perform
    - **Operands** – What data to use

Each instruction consists of an opcode + operand
  - Opcode - what to do - Add, Subtract, Divide
  - Operand - how to do it - +5, -2, shift 1
  - PowerPC – All contained within 32 bits

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operand(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r3, r10, r15</td>
</tr>
</tbody>
</table>

Simple picture of a processor architecture:
  - Expand upon it slightly as the semester goes on
  - For Cpr Es – 305 (Computer Architecture), 585 (Advanced Architecture)
Programming model includes register set and instruction set.

3.2.3 Program Execution

How does the CPU know its current position in your program?
- Internal register dedicated to telling the CPU where the next instruction is
- **PC** - Program Counter
  - PowerPC – 32 bits wide
  - Contains the memory address of next instruction
  - Must be able to address all memory
    - Memory space = 32 bits \( \rightarrow \) ~ 4 GB
- **Note**: Program code can be anywhere, CPU does not know a distinction between data portion of memory and program portion of memory
- Can we execute our data?
  - Yes - but not well, i.e., unpredictable results or invalid instructions
  - OS may add support to prevent such cases from occurring

3.2.4 Data Registers

- Need someplace to store variables internally
- Two main styles
  - Accumulator
    - Few data registers
    - Frequent swapping between stack & registers
    - Restrictions on the use of each register
  - General Purpose
    - Many registers – 8, 16, 32, 64, etc.
    - Can do any operation on any register
- **PowerPC 555**
  - General purpose processor
  - 32 GPRs (General Purpose Registers), R0..R31
    - Use notation of r0, r10, r31
    - 32 bits wide
      - To use a smaller variable type (i.e. char), only uses a portion of the register
  - 32 FPRs (Floating Point Registers), FPR0..FPR31
    - Use notation of fr0..fr31
    - 64 bits wide
      - 32 bits – float type – Single precision floating point
      - 64 bits – float type – Double precision floating point
  - Not all processors have built-in floating point support
    - Can provide support via software emulation
    - Motorola 68HC11 is an example
3.2.5 **Other Registers**
- Code maintenance
  - Stack pointer
  - Link register
- Control
  - Condition register

3.2.6 **Temporary Storage**
- What happens when we need more storage than we have registers?
  - Program with 100 variables
  - Need a place to store the values
  - Solution: Stack
- Stack
  - Temporary storage location in memory
  - Allocated block of memory
  - LIFO - Last in First Out
  - Put in values 5, 10, 15
  - Get out as 15, 10, 5, i.e. last in is the first out
  - More on stack in later lectures
- Use stack for:
  - Local variables - C functions use stack
  - Temporary storage - Calculating a long if expression or numeric value
  - Saving snapshot of variables
- Stack Pointer
  - Next location to push information onto stack (add)
  - Same size as memory space
    - PPC 555 → need 32 bits
- Some platforms have dedicated registers
  - Motorola 68HC11 – SP – Stack Pointer
  - PowerPC 555
    - No specific support for stack
    - User-selectable
    - CodeWarrior uses r1 as stack pointer

3.2.7 **Condition Register**
- Saves status of last ALU operation
- Use for program control
  - If $x > 5$, do this
- PowerPC 555 Flags
  - CR – Condition Register
  - CR0 – portion of register containing ALU status
  - Only certain instructions affect CR
    - Specified by ‘.’
• Important flag
  • EQ flag – Zero bit – Was the result zero (equal) or non-zero?

### 3.2.8 Summary of MPC555 Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>32-bit</td>
<td>Program Counter</td>
</tr>
<tr>
<td>r0-r31</td>
<td>32-bit</td>
<td>data registers</td>
</tr>
<tr>
<td>fr0-fr31</td>
<td>64-bit</td>
<td>floating point regs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR</td>
<td>32-bit</td>
<td>condition reg</td>
</tr>
<tr>
<td>CR0</td>
<td></td>
<td>ALU Integer Ops</td>
</tr>
</tbody>
</table>

### 3.3 Simplified Instruction Execution Cycle

#### 3.3.1 How does sequencing of instructions work?

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operand(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32 bits

1. Read instruction from memory (32 bits)
   Instruction = Mem[PC]..Mem[PC+3]
   PC = PC + 4  // Point to next instruction

2. Decode opcode
   What operation are we going to do (add, subtract, and)?

3. Select operands
   Which registers are we going to be using?
   Is there a value specified in the instruction?

4. Execute instruction
   Calculate with ALU (e.g., r0+5, r8 & 0x80, etc.)
   Calculate an address to read from memory, etc.

5. Writeback
   Write the result to memory or back to the register file

#### 3.3.2 Source Code
3.3.2.1 Machine Language

- Sequence of 1's and 0's executed by each machine
- Also known as Level 0 - recall levels of abstraction
- Unique to each architecture
  - Intel x86
  - Motorola 68HC11
  - PowerPC
  - StrongARM

Each instruction is executed in 3 steps
- Fetch - Get the instruction from memory
- Decode - Determine which registers, what data
- Execute - Do the instruction

3.3.2.2 Assembly Language

Assembly Language, Assembler, Assembly

An assembler translates assembly language into object code that is linked into machine code. Assembly is the process of translation.

Why learn PowerPC assembly language?
- Teach tradeoffs between C and assembly
- Fairly easy assembly language
  - Similar to MIPS, StrongARM
  - Teach assembly basics - registers, addressing modes, etc.
3.4 Instruction Set Architectures

3.4.1 CISC - Complex Instruction Set Computer

Example Processors
- Intel x86
- Motorola 68k
- Motorola 68HC11

3.4.1.1 CISC rationale

- Use microcode w/ROM
  - Recall the simple computer from Cpr E 210
  - Each machine code instruction maps to a set of microcode instructions
  - Allows for more complexity than a hardwired design
    - Can reprogram as necessary to test new designs
- Build "rich" instructions
  - Cut down on total # of instructions → smaller code size
  - Memory is expensive, try to conserve
- Build "high-level" instructions
  - High-level languages map directly to assembly opcodes
    - Support for calling C functions
    - String copy
  - Fewer lines required to translate source code to assembly code
    - Smaller code size
- CISC was a driving force behind most computers to the late 1980's

3.4.1.2 Common characteristics of CISC

- Allows multiple operands
  - 0 to N operands
- Multiple addressing modes
  - Same instruction name with different notations
    - LDAA $AA ; Load from memory
    - LDAA #$AA ; Load in value
- Variable length instructions
- Instructions take multiple clock cycles
  - Load – 3 cycles
  - Divide – 40 cycles

3.4.1.3 Advantages

- Assembly programming is easier
  - Mapping from C to assembly instructions
- Ease of upward compatibility
  - Lots of instructions to use
3.4.1.4 Disadvantages

- Complex hardware & instruction sets
  - Difficult to scale up speed-wise due to complexity
- Different instructions take different amounts of time
  - Difficult to pipeline instructions – more later on
- Many specialized instructions
  - Use only about 20% of actual instructions in a program
  - 80% of complexity in chip is not used
- Most instructions cause side effects in a CCR (Condition Code Register)
  - A “disadvantage” - more in later lectures

3.4.2 RISC - Reduced Instruction Set Computer

Brought on by advances in semiconductor technology
- Faster memory
- Faster processors

3.4.2.1 Characteristics of RISC

- Simple instruction set
- Same length instructions
- Single machine cycle instructions - allows for pipelining

Pipelining

- Executing instructions in parallel vs. sequential
- Example:
  - 5 stage processor
  - Each instruction takes 5 cycles to finish
  - Once pipeline is fully loaded, one instruction is finished per cycle
- More detail in CprE 305
• Current processors
  o AMD Athlon & Pentium
  o Use CISC external, RISC internal
  o Pentium 4 – 20-stage pipeline
  o Branch prediction
    ▪ Guessing result of if, else if
    ▪ Key to keeping pipeline full
  o More vs. fewer stages?
    ▪ More – upside: Smaller amount to do per stage, easier to turn up clock speed
    ▪ More – drawback: Difficult to keep pipeline full due to incorrect branch predictions
• Key to speed of modern processors

3.4.2.2 Advantages of RISC

• Speed increase - 2 to 4 times if properly optimized
• Simpler hardware
• Shorter design cycle because of simpler hardware

3.4.2.3 Disadvantages

• Code Quality
  o Performance of program directly correlates to quality of compiler and ability to pipeline commands
• Debugging
  o Code may not be executed as listed in source code in order to optimize execution of code
• Code Expansion
  o More instructions → large code size
• System Design
  o Requires large amounts of fast memory
  o Power consumption of higher speed processor

3.4.3 Why CISC or RISC?
• Depends upon application
  o For small or low power applications
    • Code size often outweighs code speed
    • Do not need the performance of RISC
      • 1.2 GHz Athlon in a camera?
  o For less restricted applications
    • RISC offers additional features, processing power
• More embedded systems use CISC rather than RISC
  o Battery lifetime, etc.

4 PowerPC Assembly Language

Why do we use the PowerPC and not some other processor?
• Assembly concepts are universal
  o Register, load/store, stack
  o Same concepts on x86, PowerPC, MIPS
• Many different types of assembly
  o Each is unique to each processor
• RISC programming – common across many platforms
• Can apply concepts of general purpose processor to accumulator-style

4.1 Assembly Instructions

• Fall into three basic categories
  o Register to memory
  o Memory to register
  o Register to register
  o Note: No memory to memory instructions
• RISC
  o Only load and store instructions access memory
  o Instruction mnemonic uniquely defines its operation
• CISC
  o Multiple addressing modes
  o Addition of a # may change instruction (68HC11)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDD $1808</td>
<td>D ← Mem[0x1808]</td>
</tr>
<tr>
<td>LDD #$1808</td>
<td>D ← the value 0x1808</td>
</tr>
</tbody>
</table>
4.2 Assembly Terminology

- Immediate
  - Value contained in instruction
- Indexed addressing
  - Reading to/from memory
  - Use base + offset
- Branching
  - Decide to execute an instruction based on condition flag
- Labels
  - Tell where to branch to
  - Label if/else statements, loops, functions
- Notation
  - $ is used to represent hex
    - Same as 0x
    - $80 = 0x80
    - Can use either notation in code

Example Assembly Code

```
lbz r5, 0(r10); Read the coolant temperature
li r6, CTOFFSET; Read the offset
sub r4, r5, r6; Subtract the offset
stb r4, 0(r11); Store the result back out to memory
```

- In assembly, typically each line is followed by a comment to detail what is happening.
- A comment is denoted by a “;”
- Without commenting, assembly code is extremely difficult to follow and debug.
- For lab, you should have a 1:1 or better ratio of comment to coding

4.3 Addressing Modes

4.3.1 Immediate Addressing

- Used when we want to specify a value
  - A specific number (5, -5, 3.2)
    - x = 5;
    - cChar = ‘A’;
- Tells where to find the data
  - Immediate - take the data directly from the immediate source code, i.e., take the value given
4.3.1.1 Example

\begin{verbatim}
li r5, 1500 ; r5 ← the value 1500
andi r8, r6, $80 ; r8 ← r6 & 0x80
\end{verbatim}

4.3.2 Indexed Addressing

- Works similar to array/pointer notation

Memory Address = Base + Offset

Base: Register (r0..r31)
Offset: Signed 16 bit value (-5, 25, 200)

Offset also called Displacement

- Array notation

  \begin{verbatim}
  char szArray[200]; /* Starts at 0x7E00 */
  \end{verbatim}

  szArray[10] is the same as *(szArray+10)

  Base: szArray
  Offset: 10 * sizeof(char) = 10 bytes

- Use a register as a pointer to the start of a block of memory
  - Lookup table
    - Values for keypad on PowerBox
  - Array of values
    - String, integers, longs, etc.
  - Block of memory
    - Access several memory locations based on initial value

4.3.2.1 Example

\begin{verbatim}
lbz r6, -6(r4) ; r6 ← Mem(r4 - 6)
\end{verbatim}

Base: r4
Offset: -6 bytes

4.3.3 Effective Address for a Load or Store Instruction

Effective address: resulting address after all calculations; denotes address where data is located; address of the operand
Effective Address Calculation Techniques (from Goodman & Miller):

1. The instruction might occupy two words. The first word would contain the opcode and the target register specification. The second word would contain the effective address. One way to think of this technique is that the address is a constant that happens to be stored after an instruction. Note that the hardware must know about this format so that it can adjust the PC appropriately when executing the instruction.

   ![Diagram of a two-word instruction structure]

2. The instruction might specify a register that contains the effective address, instead of specifying the address itself.

   ![Diagram of a one-word instruction structure]

This capability is important when the address itself is a variable. A load instruction given in this format would need three fields, one for the opcode, and two register specification fields. One of the register specifications would identify where the data is to go. The address of the data to be loaded is contained in a register given by the other register specification.
3. The instruction might specify a small constant and a second register. For a load instruction, the address of the data to be loaded would be obtained by adding together the contents of the register and the constant. Note that if the constant is zero, then this method of specifying an address is the same as (2) above.

4. The instruction might specify two additional registers (instead of an address). The address desired would be obtained by adding together the contents of the two registers.
Addressing Modes (from Goodman & Miller):

- Immediate
- Register
- Direct
- Register Direct
- Base Displacement (Indexed)
- Indirect

Many computers offer a variety of methods for specifying an address within an instruction. The methods are called addressing modes. The addressing mode for an operand (or the mode implied by an instruction) defines how the operand address is determined. Computers vary widely in the addressing modes they support. A few common modes are supported by nearly all computers, while others are rarely supported, particularly on load/store architectures. Computers that access memory locations for instructions other than loads and stores often include a wide variety of addressing modes. They often have variable-length instructions to accommodate the range of operand specifications. Some typical addressing modes are the following:

**Immediate**

The operand is contained directly in the instruction. Note that for a variable-length instruction, the entire operand can be included.

**Register**

The operand is contained in a register.

**Direct**

The address of the operand is contained in the instruction. Again space is required in the instruction for an entire address.
Register Direct
The address of the operand is contained in a register.

Base Displacement
The address is computed as the sum of the contents of a register (the base) and a constant contained in the instruction (the displacement). This mode is sometimes referred to as indexed or relative.
Indirect

The instruction specifies a register containing an address the content of which is the address of the operand.

```
one-word instruction
  opcode  reg
    reg   address
      memory address
        address
          effective address
```

In a load/store architecture, only two classes of instructions specify addresses: load/store instructions and control instructions. Control instructions must specify the target address of the next instruction to be executed if the transfer is taken. Because branches tend to be over relatively short distances, most target addresses can be specified as a small offset added to the address of the instruction. The address of the branch instruction is of course the value of the PC at the time the instruction is executed. Therefore the PC with a small integer offset is an efficient way to specify a target address. Branch instructions are commonly encoded this way the assembly language instruction specifies a label, such as

```
beq var1, var2, label
```

the machine language instruction specifies an offset.

```
beq var1 var2 offset
```

The offset field is calculated when the program is assembled. During execution, the offset field is sign-extended and then added to the current PC. This result gives the target address of the taken branch. That is written to the PC in the event of a taken branch. Addressing modes imply the use of the PC are termed PC-relative.
### 4.4 Labels

- Same as labels in C
- Syntax:
  - Identifier: /* Must start with a letter */
  - Label name followed by a colon
- Takes up no actual space in final code
  - Placeholder for the address of the code at that location
  - Simply tells you what the memory address of the next line of code is

**Example**

```
StartCode:
0x000020F0  li    r2, $2000
0x000020F4  addis r5, r2, $1000
AddVal:
Exit1:
0x000020F8  stw   r10, 0(r5)
```

StartCode is the same as writing the memory address 0x000020F0
AddVal and Exit1 point to the same location (0x000020F8)

- The values of the labels change as your code changes
  - Add/remove lines of code
  - As you compile, assembler updates label locations

#### 4.4.1 Labels for Functions

Start of a function/code
- The start of a function in C is just a memory address
- Same for a function in assembly
- In C, get the function address by not using parentheses
  - void DoSomething ()
- DoSomething is the address of the function

```c
void DoSomething ()
/* Located at 0x20F42386 */
{
}
```
main ( )
{
    long lFuncAddress;
    /* 32 bits - to hold memory address */

    lFuncAddress = (long) DoSomething;
    /* lFuncAddress is now equal to 0x20F42386 (the address of the function) */
}

• Why does this work?
  • PowerPC address width = 32 bits
  • long is 32 bits wide
  • Memory address is the same size as a long
  • Typecast the function to get the 32 bit address

• Review
  • Function with parentheses → calls the function
    o DoSomething ();
  • Function with no parentheses → address of function
    o DoSomething
  • Similar to array
    o Brackets versus no brackets

In assembly, the function is denoted simply by a label

4.5 Basic Assembly Instructions
  o Data Movement
  o Data Manipulation

4.5.1 Data Movement
  o Processor has finite number of registers
  o Programmer must decide what registers to use and when
    • Processor does not know what the variable nVal is
    • Only knows r4 contains 10

Typical data movement
  1. Load value into a selected register
  2. Do some sort of operation (e.g., add, and, xor)
  3. Store value back to memory

Example:

nVal = nPressure - 25;
/* 1. Load nPressure into a register */
4.5.1.1 Immediate Load

- Immediate – Load a specific value

```
char nVal;

nVal = 10;
```

- Sequence:
  1. Select a register to hold the variable (r0 .. r31)
  2. Select an appropriate instruction based on the variable type

- Immediate value 10
  - Where does 10 come from?
  - Must be specified in the program

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>li</td>
<td>Load immediate</td>
<td>Bits D16-D31 (lower half) ← IValue (16 bits)</td>
</tr>
<tr>
<td>lis</td>
<td>Load immediate shifted</td>
<td>Bits D0-D15 (upper half) ← IValue (16 bits)</td>
</tr>
</tbody>
</table>

Syntax

```
li rA, IValue ; rA ← IValue

li r3, 200 ; r3 ← 200
```

- Why is the lis instruction included?
  - RISC: fixed instruction length
PowerPC: 32-bit instruction word

What if loading a 32-bit immediate value?

Must have room for opcode
- Requires two instructions
- Load 16 bits at a time

To write the above C code example in assembly:
1. Select r4 as the destination register
2. Select li as the instruction since a char is 8 bits (8 bits < 16 bits)

\[
; \text{nVal} = 10
\]
\[
\text{li r4, 10 ; r4} \leftarrow \text{the value 10}
\]

4.5.1.2 Memory Load

- Read/write to a memory location
  - DIP switch, LED bargraph, keypad
  - Could be RAM, ROM, or a device

Consider the following code:

```c
char byValue;
char * pMemory;

pMemory = (char *) 0x100045A4;
byValue = *(pMemory);
```

- Review
  - Immediate – specify the actual value (5, -3, 200)
  - Memory load – read/write a value from/to a device

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>C Variable Type</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbz</td>
<td>Load byte with zero</td>
<td>char (8 bit)</td>
<td>Bits 24-31 = Mem(Offset+rI)</td>
</tr>
<tr>
<td>lhz</td>
<td>Load half-word with zero</td>
<td>short (16 bit)</td>
<td>Bits 16-31 = Mem(Offset+rI)</td>
</tr>
<tr>
<td>lwz</td>
<td>Load word with zero</td>
<td>long (32 bit)</td>
<td>Bits 0-31 = Mem(Offset+rI)</td>
</tr>
</tbody>
</table>

Syntax

```c
lbz rD, Offset(rI) ; rD = Mem(Offset+rI)
```

```c
lbz r4, 20(r2) ; r4 = Mem(20+r2)
```
NOTE: Offset works in bytes and is not dependent upon instruction

- With zero
  - Non-zero portions of register are zeroed
  - Why?
    - Otherwise, get a combination of old value and new value
    - Always use 32 bits
      - Up to programmer to decide what bits are important
- Memory load instruction
  - Indexed addressing
    - Effective address = rI + Offset
      - Offset is a fixed value
      - rI = register with the base address
    - Address to read/write = Base + Offset
  - Examples

  ```
  r2 = 0x00F4030A

  lbz r4, 0 (r2)

  r4 ← Mem(0+r2)
  = Mem(0+0x00F4030A)
  = Mem(0x00F4030A)

  lbz r4, -4 (r2)

  r4 ← Mem(-4+r2)
  = Mem(-4+0x00F4030A)
  = Mem(0x00F40306)

  lbz r4, 3 (r2)

  r4 ← Mem(3+r2)
  = Mem(3+0x00F4030A)
  = Mem(0x00F4030D)
  ```

- Recall C pointers

  ```
  char * pPointer;
  pPointer = (char *) 0x034500F4;

  *pPointer is the same as *(pPointer+0)
  ```

- Convert earlier C code
; pMemory = 0x100045A4
;
; Choose a register to hold pMemory
;      r6: pMemory
;
; Load upper 16 bits
; OR in the lower 16 bits

lis r6, $1000   ; r6 ← 0x10000000
ori r6, r6, $45A4 ; r6 ← 0x10000000 | (0x000045A4)

; byVal = *pMemory
;
; Choose a register to hold byVal
;      r4: byVal      r6: pMemory

; the statement can be rewritten as:
;  byVal = *(pMemory+0);
;
; pMemory points to a char type, so use byte-wise
; instruction to read a byte

lbz r4, 0(r6)   ; r4 ← Mem(r6+0)

Why use the lis and ori?
• Avoid issues with sign-extension of li
• Solution:
  o Load high-order bits into the register, padding the lower 16 bits with 0
  o OR the low-order 16 bits into the register, leaving the upper 16 bits unchanged

4.5.1.3 Memory Load – Arrays

• Very similar to pointers
• Consider the following C code

    short  nArray[20];     /* Starts at 0x2054AF00 */

    nArray[4] = 10;
    nArray[8] = 0xAF43;

• Can have the first access to array rewritten as

    *(nArray+4) = 10;

• Rewrite the code in assembly
; Load nArray into a register
;   r5 -> nArray
;
;   Load the lower 16 bits, then the add upper 16 bits
lis  r5, $2054  ; r5 = 0x20540000
ori r5, r5, $AF00  ; r5 = r5 | 0xAF00
; r5 = 0x2054AF00

; Now, r5 points to the beginning of nArray
;
; Need to be able to write (store)

4.5.1.4 Memory Store

- Store data out to memory
- Works in the same way as memory load
  - Indexed addressing
  - Mem(rI+Offset) ← value in register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>C Variable Type</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>stb</td>
<td>Store byte</td>
<td>char (8 bit)</td>
<td>Mem(rI+Offset) ← Bits 24-31</td>
</tr>
<tr>
<td>sth</td>
<td>Store half-word</td>
<td>short (16 bit)</td>
<td>Mem(rI+Offset) ← Bits 16-31</td>
</tr>
<tr>
<td>stw</td>
<td>Store word</td>
<td>long (32 bit)</td>
<td>Mem(rI+Offset) ← Bits 0-31</td>
</tr>
</tbody>
</table>

Syntax

stb  rS, Offset(rI) ; Mem(Offset+rI) = rS
stb  r3, 4(r10)      ; Mem(r10+4) = r3

NOTE: Offset works in bytes and is not dependent upon instruction

- Continue with the array example
  - r5 points to the beginning of nArray

; nArray[4] = 10;  is the same as *(nArray+4) = 10;
;
li  r8, 10  ; r8 ← 10
; Use immediate since we need the value 10
sth r8, 8(r5) ; Mem(r5+8) ← r8
; 8? 4 short variables @ 2 bytes each
- Explanation
  - li – Load Immediate
    - Can load up to 16 bits
    - short = 16 bits = OK
  - sth – Store Half-Word
    - short = 16 bits = half-word
    - Pointer math
      - Each element is two bytes wide
      - Offset works in bytes
      - Offset = sizeof(short) * 4

; nArray[8] = 0xAF43 same as *(nArray+8) = 0xAF43;
;
; Continue to use r8 as the intermediate register
li r8, $AF43 ; r8 ← 0xAF43

sth r8, 16(r5); Mem(r5+16) ← r8
 ; 16? 8 short variables @ 2 bytes each
4.5.2 Data Manipulation

4.5.2.1 Logical Operations

Bitwise operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>andi.</td>
<td>AND immediate</td>
<td>rA = rS &amp; IValue</td>
</tr>
<tr>
<td>andis.</td>
<td>AND immediate shifted</td>
<td>rA = rS &amp; (IValue &lt;&lt; 16)</td>
</tr>
<tr>
<td>ori</td>
<td>OR immediate</td>
<td>rA = rS</td>
</tr>
<tr>
<td>oris</td>
<td>OR immediate shifted</td>
<td>rA = rS</td>
</tr>
<tr>
<td>xori</td>
<td>XOR immediate</td>
<td>rA = rS ^ IValue</td>
</tr>
<tr>
<td>xoris</td>
<td>XOR immediate shifted</td>
<td>rA = rS ^ (IValue &lt;&lt; 16)</td>
</tr>
</tbody>
</table>

Syntax

andi. rA, rS, IValue
andi. r3, r3, $80 ; r3 = r3 & 0x80
ori r4, r3, $4F ; r4 = r3 | 0x4F

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>and and.</td>
<td>AND</td>
<td>rA = rS &amp; rB</td>
</tr>
<tr>
<td>or or.</td>
<td>OR</td>
<td>rA = rS</td>
</tr>
<tr>
<td>xor xor.</td>
<td>XOR</td>
<td>rA = rS ^ rB</td>
</tr>
<tr>
<td>nand</td>
<td>NAND</td>
<td></td>
</tr>
<tr>
<td>nor</td>
<td>NOR</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>NOT</td>
<td>rA = ~rS</td>
</tr>
</tbody>
</table>

Syntax

and rA, rS, rB
and r3, r2, r5 ; r3 = r2 & r5
not rA, rS
not r5, r5 ; r5 = ~r5

- Two categories
  - Immediate
    - Work with a specified value
      nVal = nVal & 0x80;
  - Register-Register
- Do ops between two registers
  \[ nVal = nVal \oplus byVal; \]

- What is the . (period) for?
  - Causes the instruction to update the condition register (CR)
  - Not all instructions can update the CR
  - Used for branching/flow control

### 4.5.2.2 Group Exercise
Suppose you are given the following C code:

```c
char * pSerialPort = (char *) 0x1048;
char nMask;

nMask = *pSerialPort & 0x80;
```

Write the assembly for this program fragment

### 4.5.2.3 Example

```c
char * pKeyboard;
char * pLED;

pKeyboard = (char *) 0x180C;
pLED = (char *) 0x18FF;

*pLED = *pKeyboard \oplus 0x0F;
*pKeyboard = *pKeyboard \& 0xF7;
```

**Solution**

```
; Select where to put the variables
;   r12: pKeyboard
;   r13: pLED

; pKeyboard = 0x180C
lis r12, $0000        ; r12 ← 0
ori r12, r12, $180C   ; r12 ← r12 | 0x0000180C

; pLED = 0x18FF
lis r13, $0000        ; r13 ← 0
ori r13, r13, $18FF   ; r13 ← r13 | 0x000018FF

; *pLED = *pKeyboard \& 0xFF
; Break down the expression
; Use r9 for *pKeyboard \& 0xFF
```
; Read from memory
lbz  r9, 0(r12)  ; r9 ← Mem(r12)
        ; r9 ← *pKeyboard

; Do the XOR operation
xori  r9, r9, $0F  ; r9 ← r9 ^ 0x0F

; Now, write the value out
stb  r9, 0(r13)  ; Mem(r13+0) ← r9
    ; Mem(pLED) ← r9
    ; *(pLED+0) ← r9

; *pKeyboard = *pKeyboard & 0xF7
;
; In C, each pointer read causes the CPU to go out and
; read memory unless otherwise specified since the CPU
; does not know if the memory location is a device or RAM.
; Use r9 again as the temporary placeholder
; Read from memory

lbz  r9, 0(r12)  ; r9 ← Mem(r9+0)
        ; r9 ← *pKeyboard

; AND using the immediate value
andi r9, r9, $F7  ; r9 ← r9 & 0xF7

; Write the value back out to memory
stb  r9, 0(r12)  ; Mem(r12+0) ← r9
    ; *pKeyboard ← r9
4.5.2.4 Math Operations

Follow same convention as logical operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Add</td>
<td>$rA = rS + rB$</td>
</tr>
<tr>
<td>addi</td>
<td>Add immediate</td>
<td>$rA = rS + \text{IValue}$</td>
</tr>
<tr>
<td>addis</td>
<td>Add immediate shifted</td>
<td>$rA = rS + (\text{IValue} \ll 16)$</td>
</tr>
<tr>
<td>sub</td>
<td>Subtract</td>
<td>$rA = rS - rB$</td>
</tr>
<tr>
<td>subi</td>
<td>Subtract immediate</td>
<td>$rA = rS - \text{IValue}$</td>
</tr>
<tr>
<td>subis</td>
<td>Subtract immediate shifted</td>
<td>$rA = rS - (\text{IValue} \ll 16)$</td>
</tr>
<tr>
<td>mullw</td>
<td>Multiply word (lower 16 bits)</td>
<td>$rA = rS * rB$</td>
</tr>
<tr>
<td>divw</td>
<td>Divide word</td>
<td>$rA = rS / rB$</td>
</tr>
</tbody>
</table>

Syntax

```
add  rA, rS, rB ; \text{rA} \leftarrow \text{rS} + \text{rB}
add  r1, r4, r10 ; \text{r1} \leftarrow \text{r4} + \text{r10}
```

4.5.2.5 Example

```c
short  nVal; /* Variable located at 0x0000FA02 */
short * pPressure; /* Variable located at 0x0000FA04 */

nVal = 0;

pPressure = (short *) 0x00002500;

nVal = (*pPressure & 0x003F) + 250;
```

**Solution**

; Select registers to use
;   r20: Memory location of nVal
;   r21: Memory location of pPressure
;   r10: nVal variable
;   r11: pPressure variable
;   r12: *pPressure

; Use registers to contain the address for indexed addressing
; of where the variables start
;
; How could only 1 register be used for accessing both variables?
lis r20, $0000  ; r20 ← 0
ori r20, r20, $FA02  ; r20 ← 0x0000FA02

lis r21, $0000  ; r21 ← 0
ori r21, r21, $FA04  ; r21 ← 0x0000FA04

; nVal = 0
li  r10, 0  ; r10 ← the value 0
sthr10, 0(r20) ; nVal ← 0

; pPressure = (short *) 0x00002500;
;
; Set the pointer to the appropriate spot

lis r11, $0000  ; r11 ← 0
ori r11, r11, $2500  ; r11 ← value 0x00002500

stw r11, 0(r21) ; pPressure ← 0x00002500

; nVal = (*pPressure & 0x003F) + 250;
;
; Break down the code into parts
; Calculate *pPressure & 0x003F
;
; Use r12 as a temporary variable

lhz r12, 0(r11) ; r12 ← value at memory location (r11+0)
; r12 ← *pPressure = *(pPressure+0)
; r12 ← Mem(0x00002500 + 0)

andi r12, r12, $003F  ; r12 ← r12 & 0x003F

; Add 250 to the expression
addi r12, r12, 250  ; r12 ← r12 + 250

; Finish by writing to nVal
st hr12, 0(r20) ; nVal ← result of the expression
### 4.5.2.6 Register-Memory Diagram of Example Solution

![Diagram]

### 4.5.2.7 Shift Operations
- Recall two types of shifts from C
  - Arithmetic, Logic
  - More on arithmetic shift later
- Logic
  - Right shift - Shift in a zero
  - Left shift - Shift in a zero

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>srwi</td>
<td>Shift right word immediate</td>
<td>( rA = rS &lt;&lt; IValue )</td>
</tr>
<tr>
<td>slwi</td>
<td>Shift left word immediate</td>
<td>( rA = rS &lt;&lt; IValue )</td>
</tr>
</tbody>
</table>

**Syntax**

```
srwi rA, rS, IValue ; rA = rS << IValue
srwi r2, r2, 5 ; r2 = r2 << 5
```
Example
C:
   nVal = nVal >> 3;

Assembly:
   Let r10 contain nVal
   srwi r10, r10, 3

4.5.2.8 Condition Register

- Affected only by instructions that have a “.”
  - Most PowerPC instructions do not affect the CR
  - Check reference manual to make sure of what is changed
- CR bits are used to provide conditions for branching
- Concerned with CR0 portion of CR (Condition Register)
  - EQ - Zero bit
  - LT - Negative bit (i.e. is MSB set)
  - GT – Positive bit (i.e. is MSB not set and not zero)

Zero Bit

- Set whenever the result of the last operation is zero
  - EQ = = 1 means that the last result was zero
- Source of Boolean true/false
  - EQ = = 0 (Boolean True - Non-Zero)
  - EQ = = 1 (Boolean False - Zero)
- Used for equality/comparison
  - How can this occur?
    - Consider the result of X - Y
    - If the result is zero, X = Y
      - 5 vs. 5
        - 5 – 5 = 0
        - EQ = 1, zero
      - 5 vs. 15
        - 5 – 15 = -10
        - EQ = 0, non-zero
  - Use the EQ bit to tell if two quantities are equal by subtracting the two quantities from one another
    - Only need to check if the result is zero

4.5.2.9 Arithmetic and the Condition Register

- Unsigned math
  - 8 bit - 0 to 255
  - 16 bit - 0 to 65,535
- Signed math
o Uses 2's complement representation
o Odometer example

-1
0xFF
0
0x00
+ Increasing value
0x80
127
0x7F
-127
0x81
-128
0x80
127
0x7F

o When MSb is 1, value is negative with a signed number:
  0xFF to 0x80 denotes –1 to –128

o To calculate the 2's complement (negation) of a signed 2’s complement number:
  Invert all bits
  Add 1 to value

  Example: Take the 2's complement of the decimal value 2

  2 = 00000010 represented as an 8-bit signed 2’s complement number
  Inverting the bits yields 11111101
  Adding 1 yields 11111110 = 0xFE

4.5.3 Assembly Flow Control

o What is flow control?
  Essentially it is skipping code that we do not want to execute, i.e. controlling the flow of
  the program
o In assembly, typically use inverted logic
  Example:

  ```assembly
  if ( nVal > 10 )
  {
    nVal--;  
  }
  ```

  o nVal-- will only be executed if the result of the if statement is true
  o Think of it from the assembly perspective
    o Skip the code if the statement is false
4.5.3.1 Branching

- Branching allows you to skip unwanted code
- May be done two ways:
  - unconditionally (think goto)
  - conditionally (e.g., if the EQ bit is set)
- Want to do a test, then branch if the condition is met
- Branching is based off of the bits in the CR

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>Branch always</td>
<td>Goto label</td>
</tr>
<tr>
<td>bne</td>
<td>Branch if not equal</td>
<td>Branch if EQ == 0</td>
</tr>
<tr>
<td>beq</td>
<td>Branch if equal</td>
<td>Branch if EQ == 1</td>
</tr>
<tr>
<td>blt</td>
<td>Branch if less than</td>
<td>Branch if LT == 1</td>
</tr>
<tr>
<td>ble</td>
<td>Branch if less than or equal</td>
<td>Branch if LT == 1 or EQ == 1</td>
</tr>
<tr>
<td>bgt</td>
<td>Branch if greater than</td>
<td>Branch if GT == 1</td>
</tr>
<tr>
<td>bge</td>
<td>Branch if greater than or equal</td>
<td>Branch if GT == 1 or EQ == 1</td>
</tr>
<tr>
<td>bt</td>
<td>Branch if true</td>
<td>Branch if EQ == 1</td>
</tr>
<tr>
<td>(same as bne)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bf</td>
<td>Branch if false</td>
<td>Branch if EQ == 0</td>
</tr>
<tr>
<td>(same as beq)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax

```
b Label               ; Goto Label
beq ElseIf1           ; Goto ElseIf1 if equal
```

4.5.3.2 Example

```c
char * pVal = (char *) 0x40041048;
if(*pVal > 10)
    *pVal = *pVal-1;
```

**Solution**

`; Select register to use`
`; r6: pVal`

CprE 211 Lecture Notes - 80 - 2001-2002
Need to compare if \(*pVal > 10\)

- Opposite of > is <=

\[*pVal <= 10\]
\[*pVal - 10 <= 0\]

- If the result of \(*pVal - 10 <= 0\), skip the statement inside the if code

Need a way to put the result in the CR

- subi instruction does not affect CR

### 4.5.3.3 Comparison

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpwi</td>
<td>Compare word immediate</td>
<td>Compute rA – IValue</td>
</tr>
<tr>
<td>cmplw</td>
<td>Compare logical word</td>
<td>Compare rA vs. rB (unsigned)</td>
</tr>
<tr>
<td>cmpw</td>
<td>Compare Word</td>
<td>Compare rA vs. rB (signed)</td>
</tr>
</tbody>
</table>

Syntax

```
cmpwi r3, 100  ; Compute r3-100
cmplw r2, r3   ; Compute r2-r3
cmpw 0, r3, r4 ; Compute r3-r4
              ; Place result in CR0
```

- Compare always affects CR
  - No “.” required
  - Can also specify which portion of CR to change
  - Performs a comparison
    - rA – IValue, rA – rB
    - Does not change actual value of register

\[\text{; Calculate } *pVal - 10\]
\[\text{; } r10 \leftarrow *pVal\]
\[\text{lbz } r10, 0(r6); r10 \leftarrow \text{Mem}(0+r6)\]
\[\text{; } r10 = *pVal\]
cmpwi  r10, 10  ; Calculate r10-10

; We want to skip the code if *pVal is not > 10
; Skip if *pVal <= 10
; <= is the opposite of >
ble  SkipIf    ; Skip to this label if
      ; *pVal <= 10

; At this point, branched if *pVal is not
greater than 10
; Thus, do the true part of the statement
; Compute *pVal - 1
lbz  r10, 0(r6)  ; r10 ← *pVal
subi r10, r10, 1  ; r10 ← r10 - 1
stb  r10, 0(r6)  ; Write back the result
      ; *pVal = *pVal - 1

; Done with the if statement

SkipIf:
      ; Other code resumes here
4.5.3.4 Longer Example

```c
if( (lVal > 42) && (byVal & 0x80)) {
    byVal = byVal ^ 0x0F;
    lVal = lVal + byVal;
}
```

Assume lVal is at 0x2584005A, unsigned long
Assume byVal is at 0x2584005E, char

Solution in assembly

```assembly
; Select registers
;   r15 points to the start of the two variables
;   r15 + 0: 0x2584005A
;   r15 + 4: 0x2584005E

lis    r15, $2584 ; r15 $high-order 16 bits
ori    r15, $005A ; r15 $low-order 16 bits

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; First task is to break down the && into separate parts. What is lazy evaluation?

; Check if lVal > 42 (opposite is lVal <= 42)

; Get the value of lVal
;   Use r8 for lVal

lwz    r8, 0(r15); r8 $Mem(0+r15)
        ; r8 $lVal
cmpwi   r8, 42 ; Is r8 <= 42?
ble     SkipEndIf ; Stop evaluation and branch to end of if statement

; At this point, know lVal > 42 since did not branch

; Check if (byVal & 0x80) is true or false
;   Use r9 for byVal

lbz    r9, 4(r15); r9 $Mem(4+r15)
        ; r9 $byVal
andi.   r9, r9, $80 ; r9 $r9 & 0x80

; Must update CR as per instruction

; Skip if statement if (byVal & 0x80) is false
; False means the result of andi is zero (EQUAL)
beq SkipEndIf

; Now, both lVal > 42 and byVal & 0x80 are true
; Reload byVal since we changed it
lbz r9, 4(r15); r9 ← Mem(4+r15) 
    ; r9 ← byVal

; Compute byVal ^ 0x0F
xori r9, r9, $0F ; r9 ← r9 ^ 0x0F
stb r9, 4(r15); byVal ← byVal ^ 0x0F

add r8, r8, r9; Add byVal to lVal 
    ; r8 ← r8 + r9

stw r9, 0(r15); Write out lVal to memory

; end of if statement
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

SkipEndIf:
; Some other code follows

○ How could this could be improved, i.e. made to run faster?
○ How would you need to change the code in both C and Assembly to test to see if both
  bits 7 and 6 are true?

4.5.3.5 Yet Another Example

Flow control: Branch instructions can change the sequence of instruction execution, i.e., skip
over code, or to go to specific code.

    char nVal;

    if (nVal > 10)
        nVal -= 3;
    else if(nVal > 5)
        nVal -= 2;
    else
        nVal += 5;

**Solution in assembly**
1. Allocate memory space for nVal in the program’s global data space: at end of
assembly code (after all instructions)

   nVal:
       .byte 0 ; Allocate a char (1 byte) with
               ; an initial value of zero

2. Select registers for the variables

    r8: address of nVal
    r10: value of nVal

3. Now, do the assembly code translation

    ; Get the memory address where nVal is at
    ; r8 ← &nVal

    lis r8, nVal@h ; r8 ← upper 16 bits of nVal’s
                    ; memory address
    ori r8, r8, nVal@l ; r8 ← r8 | lower 16 bits of
                         ; nVal’s memory address
    
    ; At this point, r8 now contains the memory address of nVal
    ; Check to see if nVal is > 10
    cmpwi r8, 10
    
    ; Skip to the else if when nVal is not > 10, i.e. nVal ≤ 10
    ble ElseIf1 ; Skip ahead if nVal ≤ 10

     IfTrue1:
     ; This label is not required but we will only get here if
     ; nVal > 10 is true since the branch is taken in all other cases

     ; nVal -= 3  Use r10 for nVal

     lbz r10, 0(r8); r10 ← Mem(r8+0)
        ; = Mem(&nVal)
        ; = nVal
     addi r10, r10, -3 ; r10 ← r10 - 3
        ; = nVal - 3
     stb r10, 0(r8); nVal = Mem(r8+0) ← r10
     b EndIf1 ; Done, skip else if, else

     ; See if the else-if condition is true

     ElseIf1:
cmpwi  r10, 5 ; Compare to see if nVal > 5
ble   Else1    ; Branch if nVal <= 5

ElseIfTrue1: ; else-if condition is true

; nVal -= 2

    lbz r10, 0(r8); r10 ← Mem(r8+0)
    addi r10, r10, -2 ; r10 ← nVal - 2
    stb r10, 0(r8); nVal = Mem(r8+0) ← r10

b   EndIf1    ; Skip over else-if, else

; Now do the else condition - executed when all other conditions fail

Else1:

; nVal += 5

    lbz r10, 0(r8); r10 ← Mem(r8+0)
    addi r10, r10, 5 ; r10 ← nVal + 5
    stb r10, 0(r8); nVal = Mem(r8+0) ← r10

; No branch required since the next line is the end of the if block

EndIf1:

; Other code will follow

4.5.3.6  Advanced Flow Control

How far can a branch go? That is, what is the range for the target address of a branch instruction?

- Two ways to calculate the target address for unconditional and conditional branch instructions
  - Displacement = number of bytes to move
  - Relative addressing: new PC = current PC + displacement (+/-)
  - Absolute addressing: new PC = displacement

- Relative addressing

Remember that an instruction word takes 4 bytes, so instruction addresses fall on word boundaries, i.e., a new instruction starts every 4 bytes. This means that an address for an instruction always has 00 for bits 0 and 1.

Simplified Example
Instruction @ Address = 0x000030D4, where 0x4=0100
Next Instruction @ Address = 0x000030D8, where 0x8=1000
Suppose an instruction word includes a 4-bit field for the branch displacement value, D.

Let effective displacement, \( ED = D \), right-padded with 00, and sign-extended as needed.

So, in this simple example, \( ED \) is a 6-bit, signed 2's complement number. The range of \( ED \) is \( 2^6 \) or 64 bytes wide. The biggest negative displacement is 100000 = \(-2^5 = -32\); and the biggest positive displacement is 011100 = +28, i.e., four-byte multiple or word-aligned value less than \( +2^5 - 1 \), which is \( +2^5 - 4 \).

This means that a branch instruction could go to code that is up to 32 bytes, or 8 instructions, behind the current PC (address of next instruction), or 28 bytes, or 7 instructions beyond.

Unconditional branching in PowerPC
- D is 24 bits
- ED is 26 bits
- Range is \( 2^{26} \) or 64 MB wide relative to PC
- Biggest negative displacement = \(-2^{25} = -32 \) MB
- Biggest positive displacement = \( 2^{25} - 4 \) = approx. +32 MB

Conditional branching in PowerPC
- D is 14 bits
- ED is 16 bits
- Range is \( 2^{16} \) or 64 KB wide relative to PC
- Biggest negative displacement = \(-2^{15} = -32 \) KB
- Biggest positive displacement = \( 2^{15} - 4 \) = approx. +32 KB
Problem: What if we want to go back to where we branched from?
    Jump somewhere and then resume where left off
Solution:
    • bl – Branch & Link
    •blr – Branch to Link Register

4.5.3.7 Link Register

- Link Register, LR: used as a link between calling programs and subroutines
- Branch to subroutine
  - Save the location of the calling program in LR
  - Return from subroutine using LR
  - Access the contents via two simplified mnemonics
    - mflr, mtlr – Move from/to Link Register

4.6 Subroutines

Starts with a label
Ends with blr – branch to link register
Similar to a function call in C

Example

```plaintext
SomeFunc:
    li    r10, $0100; arbitrary code
    li    r3, 5  ; arbitrary code
    addi   r3, r3, 5  ; arbitrary code
    stb    r3, 0(r10); arbitrary code
    blr        ; Branch to LR
```

Calling the function looks like this in the calling program:

```plaintext
li    r5, 200  ; arbitrary code
bl    SomeFunc
li    r20, 5  ; arbitrary code
```

- How does this work?
  - Return address is saved inside the link register
  - Can also use the stack
    ```plaintext
    Push PC onto stack ; Address of next instruction
    Branch to subroutine
    Pop PC from stack
    ```

- **Key Point:** The link register MUST point to the correct location when blr is executed, otherwise the code will not work.
• Register preservation
  o **Key Point:** *Don’t change register values as a result of executing a subroutine unless explicitly stated. The register values should be the same on exit from the subroutine as on entry.*
  o **Why?**
    • If you call a subroutine and the registers change, how would you know?
    • All of your subroutines should preserve registers in this course.

• How do we use variables in subroutines?
  o Set aside memory and just access variables globally
    • `.byte`
  o Allocate registers and assign appropriate values
    • r4, r5 contain the return values for the subroutine
  o Use the stack for parameters and local variables

4.6.1 **Stack**

• No explicit stack pointer register in PowerPC
  o Intel x86, Motorola 68HC11 – Explicit register for SP
  o CodeWarrior
    ▪ r1 is the Stack Pointer
    ▪ Also reference it using SP

      \[
      \text{addi SP, SP, 15}
      \]

• 32-bit memory address
• SP holds the address of the next location to push information
• Grows downward, toward lower addresses
  o Put an item on the stack
    ▪ SP decreases
  o Get an item off the stack
    ▪ SP increases
  o Start stack at a high address
    ▪ e.g., 0x2FFFFFFF
• Two basic operations
  o Push – Place/put item on the stack
    ▪ Write value to Mem(SP), decrease SP
  o Pop/Pull – Remove/get item from the stack
    ▪ Increase SP, read Mem(SP)
• Some processors have explicit stack operations
  o Motorola 68HC11
    ▪ Actual Push/Pop instructions
  o MPC 555
    ▪ Need to use store/subtract (Push) or add/load (Pull) instructions
4.6.1.1 Examples

SP = 0x7FFF

SP points to next location to push data to

Push 15 onto the stack

Takes up 1 byte on the stack
SP is now pointing to 0x7FFE (next location to push data to)

Push -3 onto stack

Takes up 1 byte on the stack
SP is now pointing to 0x7FFD
Push 0x1808 onto stack

0x1808 is a 16-bit value
Takes up two bytes on the stack
Note:
  Pushed LSB, then MSB
  Why?
    Little Endian: LSB is at higher address
    Stack grows toward lower addresses

SP is now 0x7FFB

Pop a byte value

Change SP first by incrementing, then read value
Value read = 0x18
SP is now 0x7FFB
Pop a half-word value

![Stack Diagram]

Change SP, read MSB
Change SP, read LSB
Value = MSB, LSB
0x08, –3
0x08, 0xFD
0x08FD
SP is now 0x7FFE

Notes

- When you pull an item from the stack, the value is still there in memory (i.e., it is not cleaned out)
  - Declare local variables in C, initial values are whatever was on the stack
- The stack will do exactly what you tell it
  - Does not know that a byte is part of a 32-bit, 16-bit, or 8-bit value

4.6.1.2 Push/Pop on PowerPC

- Push
  - Place a new value onto the stack
    - Write the value – stb, sth, stw
    - Decrement the stack pointer – addi or subi
  - Two-step process

Push the word in r10 onto the stack

```
stw r10, 0(SP) ; Write to the stack
addi SP, SP, -4 ; Change the value of SP
```

or

```
addi SP, SP, -4
stw r10, 4(SP)
```

- Pop
  - Remove a value from the stack
    - Increment the stack pointer – addi or subi
    - Read the value from the stack – lbz, lhz, lwz
Two-step process

Pull a value from the stack to r15

\[
\text{addi SP, SP, 4} \quad ; \text{Increment SP by 1 word (4 bytes)} \\
\text{lwz r15, 0(SP)} \quad ; \text{Read a value from the stack}
\]

or

\[
\text{lwz r15, -4(SP)} \\
\text{addi SP, SP, 4}
\]

4.6.1.3 Group Exercise

SP = 0x00007AFF

Push a byte 0xAB
Push a half-word 0xFA43
Push a half-word 0x7F43
Pop a word into r5

• What is the value of SP after the push operations?
• What does the stack look like after the push operations?
• What is the value of SP after the pop operation?
• What is the value of r5 after the pop operation?

4.6.1.4 Temporary Storage

• Stack may be used as a temporary storage space
  • Preserve values before calling a subroutine
    \[
    \text{stw r3, 0(SP)} \quad ; \text{Save r3 on stack} \\
    \text{addi SP, SP, -4} \\
    \text{bl FuncThatChangesR3} \\
    \text{addi SP, SP, 4} \quad ; \text{Retrieve r3} \\
    \text{lwz r3, 0(SP)}
    \]
  • Preserve registers in a subroutine

    \[
    \text{StartSub:} \\
    ; \text{Save r3 on stack} \\
    ; \text{Save r4 on stack}
    \\
    ; \text{Do some code with r3 and r4} \\
    ; \text{Restore r4} \\
    ; \text{Restore r3}
    \]
4.6.1.5 Stack Pointer – Review

- **SP – Stack Pointer**
  - Points to next location to write to stack
  - No explicit SP register in PPC
    - CodeWarrior uses r1
- **Stack grows downward**
  - E.g., 0x8000, 0x7FFC, 0x7FF8
  - For PPC, grows +/- 4 bytes at a time
    - Word alignment is enforced, where word boundaries have addresses with least-significant nibbles of 0, 4, 8, C
- **Push and pull/pop operations**
  - Push onto stack
  - Pull/pop from stack
- **Subroutine call and return**
  - In main, call Do_Tasks
  - In Do_Tasks, call Calc_SquareRoot
    - How do we keep track of where we are and where we came from?
      - Only one Link Register (LR)
      - Sufficient for exiting from Calc_SquareRoot back to Do_Tasks
      - How does Do_Tasks get back to main?

4.6.1.6 Nested Subroutines

**Example**

```c
main ()
{
    int j;
    Do_Tasks ();
    Do_Tasks2 ();
}

void Do_Tasks ()
{
    ...  
    fResult = Calc_SquareRoot(15);
    ...  
}

float Calc_SquareRoot (float fNum)
{
}
```

**Solution**

- Use the stack to save the return address
  - Before we call a new subroutine, push the current LR value onto the stack or
- Push the return address onto the stack

**Saving LR on the stack**

; Push the link register onto the stack

```assembly
mflr r4  ; Move from link register, r4 ← LR
stw r4, 0(SP)  ; Write to stack
addi SP, SP, -4  ; Adjust the stack pointer
```

; Now it’s okay to do bl (Branch & Link),
; which assigns a new value to LR, since the value
; needed by the current code is saved

### 4.6.1.7 Global vs. Local Variables

- Recall the difference between local & global variables in C
- Ways to declare variables
  - Allocate global space
    - In lab, used label and assembler directive `.byte` to name, allocate and initialize a global variable
    - Ex:
      ```assembly
      nVal:
      .byte
      ```
  - Use the registers
    - Ex:
      ```assembly
      r4 is dedicated to nVal
      ```
  - Allocate space on the stack
    - Ex:
      ```assembly
      nVal is at SP+10
      ```

### 4.6.1.8 Using the Stack for Local Variables

- Common in C
- To allocate, simply move the stack pointer to create space
4.6.1.9 Example

Local variables:

```c
long lVal;
char byVal;
char szName[10];
```

Total Space = 4 + 1 + 10 = 15 bytes

Decrease SP by 16 to make space

- Why 16 bytes?
  - PowerPC is word-aligned
  - Need to keep stack pointer at an even multiple of 4 bytes
  - Intel/68HC11 allows misaligned addressing
    - i.e., can read/write anywhere
  - Simply adjust stack pointer by 16 bytes

```
addi SP, SP, -16
```

- What does this do?
  - Moves SP only
  - Does not change the actual values on the stack
    - No write to memory
    - Only creates space for local variables
  - Think back to local variables
    - What happens when you do not initialize the variable?
      - Get whatever was on the stack
    - Some debuggers will initialize the values to zero
      - Visual C++ in debug mode
- CodeWarrior will not
- Visual C++ in release mode will not
  - Why not?
  - Takes time to initialize memory

Stack before allocating local variables

Stack after allocating local variables

- Exiting the subroutine
  - SP is simply moved to original location
  - No memory is cleared, SP is just moved
  - Hence you can still return a pointer to a local variable
    - Stack may overrun that local variable anytime and hence the variable may be garbage
4.6.1.10 Parameters

- Parameters to subroutines are passed in via the stack as well
- May also be saved in registers
- Convention must be supported by compiler

Example
  - Push parameters onto stack before calling the subroutine
  - In the calling program:
    - Push Parameters
    - Push Return Value (Optional)
    - bl Subroutine
    - Pull Return Value (Optional)
    - Pull Parameters

- Subroutine knows where to find the various variables
  - Offset from the stack pointer
    - E.g., return value at SP+20
  - ANSI C - must declare all variables ahead of time at beginning of subroutine
    - Dynamic creation makes compiling much more difficult
      - E.g., in C++, declare variables anywhere/anytime
      - How to manage stack?

4.6.1.11 Group Exercise

- Write a subroutine ActivateLight that reads Port A (0x1000, 8 bit). If bit 3 is set, set bits 5,6,7 of 0x1004 (8-bit port) to false. Be sure to preserve the necessary registers on the stack.
4.6.1.12 Example

```c
void Access_Serial();

void SimpleFunc ()
{
    char byVal;
    char *  pSerial;

    pSerial = (char *) 0x102F;
    if(*pSerial & 0x80)
    {
        byVal = *pSerial & 0x0F;
        *pSerial = byVal + 10;
        Access_Serial();
    }
}
```

Flowchart of subroutine execution:
SimpleFunc:    ; Create label for function

; Current stack

; Preserve register values for registers to be used
;    r3 = Start of local variables
;    r10 = pSerial
;    r11 = *pSerial, byVal

addi SP, SP, -4
stw r3, 4(SP)
addi SP, SP, -4
stw r10, 4(SP)
addi SP, SP, -4
stw r11, 4(SP)

; Save the link register
mflr r10
addi SP, SP, -4
stw r10, 4(SP)

; Allocate space for local variables
; char + char *
; 1 + 4 = 5 → 8 bytes
;
addi SP, SP, -8

; Use the concept of a frame pointer: a register used
; as a base address into the subroutine’s stack frame
; for accessing information using offsets
; r3 = Frame Pointer

mw SP, r3 ; r3 = SP
addi r3, r3, 4 ; r3 points to locals

; byVal r3+0 byte value
; pSerial r3+4 word memory address
; vs.
; byVal SP+4
; pSerial SP+8
; Space is set aside, now use the variables
; pSerial = (char *) 0x102F;
; strict C → assembly translation

    lis  r10, $0000
    ori  r10, r10, $102F

; Write to the local variable, pSerial

    stw  r10, 4(r3); pSerial: 4(r3) ← r10

; if (*pSerial & 0x80)
; Read byte value since char *

    lbz   r11, 0(r10)    ; r11 ← *pSerial
    andi. r11, $0080; r11 = r11 & 0x80
beq SkipTest ; Skip ahead if result is false

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Inside the true portion of the if statement
; Status report
;  r3 = Start of local variables
;  r10 = pSerial
;  r11 = *pSerial

; byVal = *pSerial & 0x0F;

lbz r11, 0(r10) ; re-read *pSerial
andi. r11, r11, $000F; r11 = r11 & 0x0F
stb r11, 0(r3); Write to local byVal
    ; byVal: 0(r3) ← r11

; *pSerial = byVal + 10;
    addi r11, r11, 10 ; Add 10 to the value

; What is the difference between this statement
stb r11, 0(r10)
    ; and
;
    ; stb r11, 4(r3)

; Call the Access_Serial function
bl Access_Serial

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Done with if statement, now finish the function

SkipTest:
; Done with function, first clean up local variables
    addi SP, SP, 8 ; Move the stack pointer
; Now restore the registers that we used
;   r3 for a frame pointer
;   r10 for pSerial
;   r11 for temporary values
;   LR is also saved but not changed

; Pull in the opposite order than pushed
;   LR, r11, r10, r3

lwz r11, -4(SP)
addi SP, SP, 4
mtlr r11
```
lwz r11, -4(SP)
addi SP, SP, 4

lwz r10, -4(SP)
addi SP, SP, 4

lwz r3, -4(SP)
addi SP, SP, 4
```

; All done - now just exit the function
blr

**Review**

- What does the example illustrate?
  - Allocating local variables
  - Preserving of registers
  - Pushing/Pulling for temporary storage
    - Always pull what you push
    - Pull in opposite order of pushing (LIFO)
  - Strict C to Assembly translation
- Why might the compiler update local variables with intermediate values in cases where it knows final values?
  - Assembly can be used to optimize code
- Why not just make all variables global variables?
  - Problem: Need to allocate space for all variables from all functions
- Why should we preserve the registers?
  - Optional when speed/size is critical
    - May end up unnecessarily pushing/pulling from stack
  - Interrupts - Always preserve registers
5 I/O Subsystems

- Transfer real-world values between system and its environment
  - Detect a temperature and adjust the climate control
  - Open a value from 5% to 20% over 5 seconds
  - Provide traction control through braking system using sensors
- Cover I/O subsystems for remainder of semester
  - Directly read/write to real-world
- Two methods to interface with external devices
  - Polling
    - CPU repeatedly reads device to check if it needs service
    - Requires some type of status to read
      - Status Register - Tells when the device needs service, i.e. a flag (bit) is set
  - Interrupt
    - Device informs the CPU that it needs service
    - Interrupt is triggered when flag is set and active until flag is cleared
- Register refers to the registers for the device
  - Specific to a device
    - Control, Status, Data
  - Mapped to memory addresses
    - E.g., PACTL = $1026, PITC = $2FC244
  - Visible to full system
  - CPU registers – not memory-mapped
    - r1, LR, CR, MSR
- Control register
  - Set rate, mode, enable interrupts, etc.
  - Examples:
    - Single conversion on ADC
    - 9,600 baud on the serial port
- Status register
  - Flags denoting status of device
  - Examples:
    - Is there a byte available at the serial port to read?
    - Is the A/D conversion complete?
- Data register
  - Actual data from I/O subsystem
  - Examples:
    - Value from the ADC (0 to 1023)
    - Value from the serial port
5.1 **A/D I/O Subsystem**

- **A/D** - Analog to Digital – also known as ADC (Analog to Digital Converter)
  - Pressure (psi): convert to an integer
    - 10.0 to 100.0 psi becomes 0 to 1023
  - Current (mA): convert to an integer
- **D/A** - Digital to Analog - also known as DAC (Digital to Analog Converter)
- Typical application for ADC’s is in a control system
- PowerPC includes an ADC on chip
  - Part of the microcontroller functionality

### 5.1.1 Converting Analog to Digital

- Analog input signal

![Analog Input Signal](image)

- Sampling over time (discretization)

![Sampling Over Time](image)

- Digitizing (quantization)

![Digitizing (Quantization)](image)
• Limitations
  o Inherent error due to discrete sampling
    • Square bars versus smooth (curved) waveform
    • Limited by speed of ADC, processor
  o Accuracy varies due to bit resolution
    • 4.12 V may appear the same as 4.16 V
    • More bits = finer granularity = better accuracy = more expensive
    • 8 bit ADC: 256 possible values can be represented

5.1.2 Terminology and Equations

Variables and equations for ADC operation:

• **Offset**: minimum analog value
• **Span** (or **Range**): difference between maximum and minimum analog values
  o Max - Min
• **n**: number of bits in digital code (sometimes referred to as n-bit resolution)
• **Bit Weight**: analog value corresponding to a bit in the digital number
• **Step Size** (or **Resolution**): smallest analog change resulting from changing one bit in the digital number, or the analog difference between two consecutive digital numbers; also the bit weight of the LSb
  o Span / 2^n

Let AV be Analog Value; DN be Digital Number:

AV = DN * Step Size + Offset = DN / 2^n * Span + Offset

DN = (AV - Offset) / Step Size = (AV - Offset) * 2^n / Span

5.1.3 ADC Implementation

• Successive Approximation
  o Common, moderate speed, moderate accuracy
• Other types: Integration, Flash, etc.

5.1.3.1 Successive Approximation

• Bit Weight concept
  o Think of each bit in the digital number as corresponding to an analog value
Example: 6-bit ADC with 0 V to 5 V range
- Bit 5 (MSb) = 1/2 * 5 V = 2.5 V
  - Therefore, when bit 5 is true, it represents 2.5 V
- Bit 4 = 1/4 * 5 V = 1.25 V
- Bit 3 = 1/8 * 5 V = 0.625 V

- Used by most ADCs
- Uses a voltage comparator
  - 1 - High (Vin > Vout)
  - 0 - Low (Vin <= Vout)

Example – 6-bit A/D
- Input voltage = 4.16
- Range = 0 to 5 V
- Cout = Comparator
- Sequence of events
  - Set bit 5 to 1 for ADC
  - Compare voltage
  - If Vout < Vin, keep bit as a 1
  - Otherwise, clear bit to a 0

<table>
<thead>
<tr>
<th>Digital Number</th>
<th>Vout</th>
<th>Vin</th>
<th>Cout Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>2.5</td>
<td>4.16</td>
<td>1 (lower)</td>
</tr>
<tr>
<td>110000</td>
<td>3.75</td>
<td>4.16</td>
<td>1 (lower)</td>
</tr>
<tr>
<td>111000</td>
<td>4.375</td>
<td>4.16</td>
<td>0 (higher)</td>
</tr>
<tr>
<td>110100</td>
<td>4.063</td>
<td>4.16</td>
<td>1 (lower)</td>
</tr>
<tr>
<td>110110</td>
<td>4.22</td>
<td>4.16</td>
<td>0 (higher)</td>
</tr>
<tr>
<td>110101</td>
<td>4.141</td>
<td>4.16</td>
<td>1 (lower)</td>
</tr>
</tbody>
</table>

- Starts with larger values, gradually approximates smaller and smaller values
- Closest value approximated by 6-bit ADC is 4.14

5.1.4 ADC on the PowerPC
• QADC64 - Queued Analog to Digital Converter Module-64
  o 16 analog channels via internal multiplexing
  o 10-bit ADC resolution
    ▪ Converts voltage to an integer value (0-1023)
  o Polling or interrupt driven
  o Programmable channels
    ▪ AN0-ANx
    ▪ More channels available (some are multiplexed)

5.1.4.1 Lab Setup

• 4 ADC channels (AN0-AN3)
• Uses a POT (Potentiometer) to supply an analog signal
  o 0-5 V through a variable resistor

5.1.4.2 Using the QADC64

1. Setup
  • Configure the ADC subsystem
    o Which channels?
    o Indication that conversion complete?
      • Polling or interrupts
    o Type of scanning?
      • Once, repeatedly, etc.
    o Precision of a reading?
2. Monitoring
   - Read the data if data at the ADC are ready to be processed
   - Convert the data from an integer (digital) value to the appropriate value
   - Enable the next scan(s)

5.1.4.3 Queued ADC

- Table of 64 entries – called Conversion Command Word (CCW) Table
  - Central element for control of the QADC64 conversions
  - Each CCW specifies the conversion of one input channel
    - Entry in table is a command word for describing a scanning/conversion operation
  - CCW Table is at address 0x304A00

- CCW Table partitionable into Queue 1, Queue 2, and subqueues for arranging and controlling more complex scan sequences
  - A queue is a scan sequence of one or more input channels
  - Each queue can be operated using several different scan modes
  - A queue is started by a trigger event, which is a way to cause the QADC64 to begin executing the command words

- Results from a sequence of conversions are placed in the Result Word Table
  - Result Word Table is at address 0x304A80

- CCW entry fields
  - Each CCW requests the conversion of an analog channel to a digital result. The CCW specifies the analog channel number, the input sample time, and whether the queue is to pause after the current CCW.
    - P – Pause after conversion until next trigger
    - BYP – Bypass buffer amplifier (affects the timing)
    - IST – Input Sample Time
    - CHAN – Channel to sample (0-63)
      - All 1’s – End of Queue

- IST – Input Sample Time
  - The IST field specifies the length of the sample window. Longer sample times permit more accurate A/D conversions of signals.
    - Longer time → More time for successive approximation
  - 00 – QCKL period x 2, 01 – QCKL period x 4, 10 – QCKL period x 8, 11 – QCKL period x 16
**CCW Table**

<table>
<thead>
<tr>
<th>Entry</th>
<th>16 bits</th>
<th>0x304A00</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Begin Queue 1</td>
<td>0 5 6 7 8 9 10 15</td>
</tr>
<tr>
<td>n</td>
<td>End Queue 1</td>
<td></td>
</tr>
</tbody>
</table>
• Total conversion time
  o Consists of initial sample time, final sample time, and resolution time
    ▪ Initial sample time – time during which the selected input channel is
driven by the buffer amplifier onto the sample capacitor (disabled by
means of the BYP bit in the CCW)
    ▪ Final sampling period – time to set up DAC array
    ▪ Resolution period – time to convert voltage in the DAC array to a digital
      value

Why Queue?

• Select desired behavior
  o Scan a few channels quickly
  o Scan a channel multiple times (change/errors)
  o Scan large number of channels

5.1.4.4 Programming the ADC

  ▪ QACR1 – QADC64 Control Register 1
    o 16 bit, 0x30480C
    o SSE1 – bit 2 – Single Scan enable
    o MQ1 – bits 3-7
      ▪ Set to binary 00001 (Queue 1)
  ▪ QASR0 – QADC64 Status Register 0
    o 16 bit, 0x304810
    o ADC sets a flag when the conversion is done
    o CF1 – bit 0 – Conversion Complete
  ▪ CCW Table
    o 16 bit, 0x304A00 start address
  ▪ Result Word Table
5.1.4.5 Example

- Configure ADC to read a value from AN0 in single channel, single read mode
- Read in a temperature from a temperature sensor that records 30 to 120 Celsius and outputs 0 to 5 V

**Step size for voltage**

Offset = Min = 0 V  
Span = Max - Min = 5 - 0 = 5 V

Step Size =

**Step size for temperature**

Offset = Min = 30 degrees  
Span = 120 - 30 = 90 degrees

Step Size =

**Temperature calculations**

Digital reading of 150 – What is the temperature?

Digital reading of 500 – What is the temperature?

5.2 Interrupts

5.2.1 Terminology

- **Exception** – Unexpected events generated internally within processor  
  - Arithmetic overflow  
  - System call  
  - Undefined instruction
- **Interrupt** – Events generated outside of processor  
  - I/O device  
    - A/D interrupt when conversion is complete  
    - Serial interrupt when a data byte is received  
  - **Interrupt Service Routine (ISR)** - Responsible for servicing interrupt
5.2.2 **Interrupt vs. Polling**

- **Polling**
  - Current method used in lab
  - Ask the device if it has any data
  - All intelligence is in the CPU
    - CPU controls interaction with the device
- **Interrupts**
  - Device tells the CPU that is ready with/for data or needs service by interrupting the CPU
  - Intelligence is moved to the device
    - Device informs the CPU when it needs service
- **Which is better/faster?**
  - Polling is simple
    - Query each device via pointers and do whatever is required
    - Very little setup required
  - Interrupts are faster
    - What happens when the device does not need service?
      - Wasted CPU cycles checking a device
      - Do you have data?
        - Yes or No
        - What if 90% of time device doesn’t need service?
    - What happens when there are many devices?
      - 10, 100, 1000 devices
      - As the number of devices increases, more and more time is spent polling a device
      - Very costly for real-time operations
  - Consider Lab 8
    - Instead of doing a busy wait for data, let the serial port tell you when data is ready to read or there is room to write
    - Allows time for other tasks (e.g., A/D) when device does not need service
  - Interrupts are more complicated
    - Difficult to debug due to asynchronous behavior
    - Debugging can alter the behavior of the interrupt
      - Transient bugs - debug version works, normal version does not
    - More setup involved to write ISR
    - All code called by ISR must be re-entrant, i.e., callable while it is already being executed
- Example of interrupt versus polling
  - Lecture
    - Polling – asking if there are any questions, students respond
      - 10 students – Ask each student
100 students – Ask each student

- Interrupt – Students raise hand, instructor calls on students

5.2.3 More on Interrupts

- Interrupt routines are usually set up to run as quickly and efficiently as possible
- Essentially, the execution of other code stops while the interrupt is processed
- Consider this sequence of events:
  - A user presses a key on the keyboard
  - An interrupt request is sent from the keyboard to the CPU
  - The CPU interrupts what it is doing to run the keyboard Interrupt Service Routine
  - Execute ISR to handle the keystroke
  - The CPU returns back to the code it was executing
- When can interrupts happen?
  - Any time in our normal code
  - Restricted cases if we are already handling another interrupt
- Reentrancy
  - An interrupt can be called at any time, therefore any subroutine that is called inside of an interrupt must be reentrant.
  - Example:
    - A subroutine is used to calculate the square root. A main program calls the subroutine. An interrupt happens, and the ISR calls the same square root subroutine, thus effectively calling the square root function again while the main code is already in the subroutine.
    - How is that possible?
      - Interrupts can happen anytime in our code
      - A device is triggering the interrupt, not the processor

5.2.4 Different Types of Interrupts

- Hardware & Software interrupts
  - Hardware – Device-driven
    - Serial port interrupts the CPU
    - A/D is done with a conversion
  - Software – called by program
    - Used by OS, debugging, etc.
    - System call
- Reset
  - Special kind of interrupt, resets to an initialization routine but does not return like a normal interrupt
  - Reset button on PowerPC
    - Raises reset interrupt for CPU
    - CPU executes reset routine
    - Memory is not cleared, only registers are initialized
      - Can press reset but do not have to download program again
5.2.5 **Vectors**

- **Vector** contains the start address of an ISR for a specific interrupt
  - What do I do if an interrupt goes off?
- **Vector table**
  - Contains a table of all vectors for all interrupts for the system at a set location
  - Different processors may or may not have vector tables
    - Motorola 68HC11, 68332
      - Yes
    - Power PC 555
      - No – Exception-based

5.2.6 **More on Exceptions**

- Exception – An out-of-the-ordinary event has occurred that requires the attention of the CPU
  - Typically occurs from an internal event
- Interrupt – External interrupt is a subset of exceptions for the PowerPC
- Exception table
  - Chapter 6 – RCPU Reference Manual (white book)
  - May be located at either 0x000xxxxx or 0xFFFxxxxx

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Vector Offset</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Reset</td>
<td>0x00100</td>
<td>External reset pin is asserted</td>
</tr>
<tr>
<td>Machine check</td>
<td>0x00200</td>
<td></td>
</tr>
<tr>
<td>External Interrupt</td>
<td>0x00500</td>
<td>External interrupt is asserted (IRQ pin)</td>
</tr>
<tr>
<td>Alignment</td>
<td>0x00600</td>
<td>Misalinged addressing used</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System call</td>
<td>0x00C00</td>
<td>System call instruction is executed (sc instruction) Used by OS</td>
</tr>
</tbody>
</table>

5.2.7 **Interrupt Handling**

What happens when an interrupt goes off?

1. Device raises the interrupt line to the CPU
2. CPU finishes its current instruction
3. External interrupt exception handler saves critical registers (CR, etc.) to stack
4. External interrupt exception handler determines which device raised the external interrupt (A/D, Serial, Reset, etc.)
5. Exception handler jumps to the appropriate ISR
6. CPU executes the ISR for the device
7. ISR concludes and returns to external interrupt exception handler
8. External interrupt handler restores critical registers (CR, etc.) from the stack
9. External interrupt handler returns with rfi instruction
10. CPU resumes normal execution

- **Key point** – to the code, it cannot tell that the interrupt ever happened unless a global flag is set
  - Essentially, the CPU stops what it is doing, services the interrupt, and resumes what it was doing
- Once interrupts are enabled, they can happen at any time until they are disabled

### 5.2.8 Interrupts and the Stack

- Since an ISR will typically want to change the contents of the registers, it will want to save a snapshot of the values in all of the registers
- Some processors will save the registers automatically
  - 68HC11, etc.
- Some will not
  - PowerPC 555, Intel x86
- Why is that?
  - PowerPC 555 has 32 registers, 32 floating point registers, other registers
  - Pushing 64 values onto the stack out to memory
  - 64 writes/reads to/from memory
  - Slows down handling of exceptions/interrupts considerably
  - Optimize code
    - Selectively preserve registers
- Do we need to preserve registers in an ISR?
  - Yes – it is not done automatically

### 5.2.9 Interrupt Activation

- Most interrupts support the concept of masking
  - Maskable interrupt – the CPU can pay attention to or ignore the interrupt
- Masking is accomplished through the use of status flag and interrupt enable bits
  
  \[
  \begin{align*}
  \text{if (status flag is set & interrupt enable)} \\
  \text{trigger/activate the interrupt}
  \end{align*}
  \]
  - If the interrupt enable bit is not set, the interrupt is masked out
- Interrupt service routine (ISR) must reset the interrupt
  - Status flag must be cleared by the ISR (it is not cleared automatically)
  - What happens if the flag is not cleared?
    - Request remains pending and interrupt triggers again
    - Why?
      - The only test to check for the interrupt is if the flag and the interrupt enable bit are set
      - If the flag is not cleared, the device still asserts that it needs service
5.2.9.1 I/O Subsystem Setup

- Done once before the subsystem is used

<table>
<thead>
<tr>
<th>Polling</th>
<th>Interrupt-Driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Configure device – rate, mode, etc.</td>
<td></td>
</tr>
<tr>
<td>2. Clear status flag</td>
<td>1. Set up ISR in vector table, exception handler</td>
</tr>
<tr>
<td></td>
<td>2. Configure device – rate, mode, etc.</td>
</tr>
<tr>
<td></td>
<td>3. Clear status flag</td>
</tr>
<tr>
<td></td>
<td>4. Enable interrupts for device, processor</td>
</tr>
</tbody>
</table>

5.2.9.2 I/O Subsystem Operation

- Done while the subsystem is used (maintenance)

<table>
<thead>
<tr>
<th>Polling</th>
<th>Interrupt-Driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Check device status flag for data</td>
<td></td>
</tr>
<tr>
<td>2. Read/write/process data</td>
<td>1. Device notifies via interrupt</td>
</tr>
<tr>
<td>3. Clear status flag</td>
<td>2. Read/write/process data</td>
</tr>
<tr>
<td></td>
<td>3. Clear status flag</td>
</tr>
</tbody>
</table>

What is the setup in lab?
- Exception handler is already written
- Write the ISR code and the main C code
- Exception handler takes care of branching/jumping to our ISR when an external interrupt exception occurs

5.3 Periodic Interrupt Timer (PIT)

- Periodic interrupt derived from a clock
  - May be system clock or external clock
  - Counts down from X to zero
    - PITC (PIT Count) – Value to count down from (i.e., X)
    - PITR (PIT Register) – Current value of the counter
      - Count down until PITR = 0
      - Set a flag
      - Set PITR to PITC and continue counting
- Discussed in more detail in MPC555 User’s Manual (Chapter 6, page 6-15)
The timeout period is calculated as:

\[
PIT_{\text{period}} = \frac{PITC + 1}{F_{\text{pitrclk}}} = \frac{PITC + 1}{\text{ExternalClock}/4\text{or}256}
\]

Solving this equation using a 4-MHz external clock and a pre divider of 256 gives:

\[
PIT_{\text{period}} = \frac{PITC + 1}{15625}
\]

This gives a range from 64 microseconds, with a PITC of 0x0000, to 4.19 seconds, with a PITC of 0xFFFF. When a 20-MHz crystal is used with a pre divider of 256, the range is between 12.8 microseconds to 0.84 seconds.

**6.13.4.9 Periodic Interrupt Timer Count Register (PITC)**

The PITC register contains the 16 bits to be loaded in a modulus counter. This register is readable and writable at any time.

| MSB | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
--- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
PITC | | | | | | | | | | | | | | | | |
RESET: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Table 6-19 PITC Bit Settings**

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:15</td>
<td>PITC</td>
<td>Periodic interrupt timing count. This field contains the 16-bit value to be loaded into the modulus counter that is loaded into the periodic timer. This register is readable and writeable at any time.</td>
</tr>
<tr>
<td>16:31</td>
<td>—</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
6.13.4.10 Periodic Interrupt Timer Register (PITR)

The periodic interrupt register is a read-only register that shows the current value in the periodic interrupt down counter. Read or writing this register does not affect the register.

**PITR — Periodic Interrupt Timer Register**

```
<table>
<thead>
<tr>
<th>MSB</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
```

**PIT**

RESET:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

**RESERVED**

RESET:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

**Table 6-20 PIT Bit Settings**

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:15</td>
<td>PIT</td>
<td>Periodic interrupt timing count — This field contains the current count remaining for the periodic timer. Writes have no effect on this field.</td>
</tr>
<tr>
<td>16:31</td>
<td>—</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
5.3.1 Configuring the PIT for Interrupt Operation

- Set up ISR in vector table/exception handler
  - Examine later
- Configure the device
  - PISCR register – Periodic Interrupt Status & Control Register
    - p. 6-32 of MPC555 manual
    - Rate – How fast does the PIT count and cycle through zero?
      - PITC – 16-bit counter – counts down to zero
      - Counts at a specified rate
      - PIT Clock Rate = Sys. Clock / Pre-Divider
  - Lab settings
    - 20 MHz System Clock
    - Pre-Divider of 4
    - PIT Period = (PITC+1) / (Sys. Clock / Pre-Divider)
    - PIT Period = (PITC+1) / (20,000,000 / 4) sec.
    - PIT Period = (PITC+1) / (5,000,000) sec.
  - Example – 1 ms periodic interrupt
    - 0.001 sec = (PITC+1) / (5,000,000)
    - PITC = 4999
  - Enable the device to count
    - Set PTE – Periodic Timer Enable – bit to 1

---

### Table 6-18 PISCR Bit Settings

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:7</td>
<td>PIRQ</td>
<td>Per router interrupt request. These bits determine the interrupts of the PIT. Refer to 6.4 Interrupt Controller for interrupt level encodings.</td>
</tr>
<tr>
<td>8</td>
<td>PS</td>
<td>Periodic interrupt status. This bit is set if the PIT issues an interrupt. The PIT issues an interrupt after the modulus counter counts to zero. PS can be negated by writing a one to it. A write of zero has no affect.</td>
</tr>
<tr>
<td>9:12</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>13</td>
<td>PIE</td>
<td>Periodic interrupt enable. If this bit is set, the time base generates an interrupt when the PS bit is set.</td>
</tr>
<tr>
<td>14</td>
<td>PITF</td>
<td>PIT freeze. If this bit is set, the PIT stops while FREEZE is asserted.</td>
</tr>
</tbody>
</table>
| 15     | PTE  | Periodic timer enable
  - 0 = PIT stops counting and maintains current value
  - 1 = PIT continues to decrement |
Clear status flag

- Why clear the status flag?
  - Ensure that a new count causes the flag to be set, and that an old value from a previous count is not re-read
- Write a 1 to the PS bit of the PISCR
  - Why write a ‘1’, shouldn’t it be a zero?
  - Timer I/O uses a ‘1’ to clear the bit

Enable the interrupt at two levels

- Device level: enable the interrupt at the PIT
  - Select the interrupt level
    - From Interrupt Level 0 to Interrupt Level 7
    - Affects the priority of the interrupt
  - Tell the device it is OK to interrupt the CPU
    - Set the PIE – Periodic Interrupt Enable – bit to a 1
- System level: allow the system to see the interrupt
  - Set the SIMASK mask register to enable the appropriate level
    - If the device is at Interrupt Level 4, interrupt level 4 should not be masked off
  - Let the processor see the interrupt
    - External interrupts can be enabled/disabled
    - Controlled in the MSR – Machine State Register
      - Refer to the RCPU Programmer Manual (p. 2-14)

![Figure 2-12. Machine State Register (MSR)](image)

Table 2-8 shows the bit definitions for the MSR.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-12</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>POW</td>
<td>Power management enable&lt;br&gt;0: Power management disabled (normal operation mode)&lt;br&gt;1: Power management enabled (reduced power mode)&lt;br&gt;Note: Power management functions are implementation-dependent. If the function is not implemented, this bit is treated as reserved.</td>
</tr>
<tr>
<td>14</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>ILE</td>
<td>Exception little-endian mode. When an exception occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the exception.</td>
</tr>
<tr>
<td>16</td>
<td>EE</td>
<td>External interrupt enable&lt;br&gt;0: While the bit is cleared, the processor delays recognition of external interrupts and decrementer exception conditions.&lt;br&gt;1: The processor is enabled to take an external interrupt or the decrementer exception.</td>
</tr>
</tbody>
</table>
### 5.3.2 Writing the ISR

- Write the code for the ISR
  - ISRs are always void functions
    - Cannot return anything since do not return anywhere specifically
  - **Key point:** A program does not call an ISR; it is called for the program by the exception handler when the interrupt occurs.
  - Two main tasks of an ISR
    - Do something related to the interrupt, e.g., process data
    - Clear status flag

- PIT ISR
  - Do something
    - No data to read
    - However, there is a task to do
      - An interrupt means that X ms have passed since the last time the interrupt was triggered
      - Hence, adjust the time appropriately or signal a global flag
  - Clear status flag
    - Covered earlier
    - Write a 1 to the PS bit of the PISCR

### 5.3.2.1 Lab Code

```c
// Interrupt_Hdlr.c : Source file for C-based interrupts
#include "Defines.h"
char * p_g7Seg = (char *) IO_DIGITAL_OUTPUT_7SEG;
char gVal = 0;
int gSlowVal = 0;

void InterruptHandlerIRQ0 (void)
{
}
void InterruptHandler0 (void)
```

CprE 211 Lecture Notes - 124 - 2001-2002
void InterruptHandler4 (void)
{
    short * pPISCR = (short *) 0x002FC240;
    long * pPITC  = (long *) 0x002FC244;
    long * pPITR = (long *) 0x002FC248;
    long * pSIMASK = (long *) 0x002FC014;
    short nPISCR;

    // Read the divider from DIPSwitch 1

    // 1. Do the task associated with the interrupt
    // - Increment the global value

    gSlowVal++;
gVal++;

    *p_g7Seg = gVal;

    // 2. Reset the flag for the interrupt

    nPISCR = *pPISCR;  // Read the PISCR

    nPISCR = nPISCR | 0x0080;  // OR in a 1 to set the bit

    *pPISCR = nPISCR;  // Write the value back out
    // A '1' resets the PS bit of the PIT subsystem
}

Do you ever call an ISR?
  o No – An ISR is called automatically by the CPU through the exception handler/vector table
  o Device interrupts the CPU
  o CPU executes an exception handler
  o Exception handler calls ISR
  o ISR returns via rfi (Return From Interrupt)

**Group Exercise**

What would happen if your main program calls an ISR?
5.4 MPC 555 Interrupt System

5.4.1 Initialization Steps

Each interrupt source must be initialized before all interrupts can be enabled in the machine state register, EE bit.

Initialization consists of four steps: module specific initialization, level assignment, enabling the interrupt source, and setting the interrupt mask in the SIU interrupt controller.

5.4.1.1 Step 1: Module Specific Initialization

Each interrupt source will need to have its own general initialization of its module.

Examples of some module specific initializations:
• Interrupt Pins: specify edge or level detection
• Timers: specify clock input selection, clock prescaler value, pre-loading value
• Serial I/O: specify baud rate, queue management parameters
• QADC: specify queue management parameters
• TPU, MIOS: specify function assignment, function specific parameters

5.4.1.2 Step 2: Level Assignment

• Lower level numbers have higher priority
• External interrupt pins do not have level assignments but have a fixed priority
• To reduce latency, each interrupt source should be mapped to its own level if possible

5.4.1.3 Step 3: Enable Interrupt

Each interrupt source other than IRQ pins must be enabled using an enable control bit for the source.

5.4.1.4 Step 4: Set Appropriate Mask Bits in SIMASK

All appropriate USIU interrupt controller levels 0:8 must have their mask bits set (enabled) in the SI-MASK register.

5.4.1.5 Final Step: Setting MSR[EE] and MSR[RI] Bits

After all the interrupt sources have been initialized to the previous steps, the enable external interrupts [EE] bit must be set for interrupts to be recognized, and recoverable interrupt [RI] set to tell exceptions the state is recoverable. Writing any value to the EIE register sets both the MSR[EE] and MSR[RI] bits. Writing is accomplished by using the mtspr instruction.

An interrupt is recognized if:

Table 9 USIU Interrupt Level Assignments

<table>
<thead>
<tr>
<th>Level Assignment</th>
<th>Binary Value</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000000</td>
<td>0x80</td>
</tr>
<tr>
<td>1</td>
<td>01000000</td>
<td>0x40</td>
</tr>
<tr>
<td>2</td>
<td>00100000</td>
<td>0x20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>00000001</td>
<td>0x01</td>
</tr>
</tbody>
</table>
### Table 16 Interrupt Event Sequence

<table>
<thead>
<tr>
<th>System Behavior</th>
<th>Software Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception occurs</td>
<td></td>
</tr>
<tr>
<td>Currently executing instructions are completed</td>
<td></td>
</tr>
<tr>
<td>The CPU saves the address of next instruction and MSR[16:31] in SRR0:1, then modifies MSR (see 3.1 PowerPC Core Interrupt)</td>
<td>1. Save &quot;Machine Context&quot; of SRR0:1.</td>
</tr>
<tr>
<td></td>
<td>2. Set MSR[RI] to indicate the state is now recoverable. Other maskable interrupts/exceptions could now be enabled.</td>
</tr>
<tr>
<td></td>
<td>3. Save other appropriate context (registers).</td>
</tr>
<tr>
<td></td>
<td>4. Determine interrupt source.</td>
</tr>
<tr>
<td></td>
<td>5. Branch to interrupt handler and execute it. If necessary, negate the interrupt request in the handler.</td>
</tr>
<tr>
<td></td>
<td>7. Return to program by executing &quot;rfi&quot; instruction.</td>
</tr>
<tr>
<td>The CPU restores return address, original MSR, and enables interrupts again.</td>
<td></td>
</tr>
<tr>
<td>Program execution resumes in the routine that was interrupted.</td>
<td></td>
</tr>
</tbody>
</table>

### 5.4.2 Initialization Code

```assembly
; Interrupt.s : Interrupt handler for the PowerPC 555
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Version: 1.00, 15/10/1999
; Author: Philippe TECHER, Virtual Micro Degin (ptecher@vmdesign.com)
;
; Source: http://www.cim.mcgill.ca/~jasmith/docs/mpc500-interrupthandler.txt
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Import the function definitions for the linker

.import InterruptHandlerIRQ0
.import InterruptHandler0 ; 1 Level 0x80 (0)
.import InterruptHandlerIRQ1 ; 2
.import InterruptHandler1 ; 3 Level 0x40 (1)
.import InterruptHandlerIRQ2 ; 4
```
.import InterruptHandler2      ; 5   Level 0x20 (2)
.import InterruptHandlerIRQ3   ; 6
.import InterruptHandler3      ; 7   Level 0x10 (3)
.import InterruptHandler4      ; 8
.import InterruptHandlerIRQ4   ; 9   Level 0x08 (4)
.import InterruptHandler5      ; 10
.import InterruptHandler5      ; 11 Level 0x04 (5)
.import InterruptHandlerIRQ6   ; 12
.import InterruptHandler6      ; 13 Level 0x02 (6)
.import InterruptHandler7      ; 14
.import InterruptHandler7      ; 15 Level 0x01 (7)

.export InitInterrupt

.function "ExceptionHandler", Start_IHdlr, End_IHdlr-Start_IHdlr
.function "InitInterrupt", InitInterrupt, End_InitInterrupt-InitInterrupt

A_INTBASE       .equ 0xFFF00500   ; Base address for EXTERNAL INTERRUPT
A_SIUBase       .equ 0x002F  ; base to access SIU Module
A_SIUOffset     .equ 0xC000  ; Offset to add to obtain SIU base address
A_SIUMCR      .equ 0x0000  ; Section 6.13.1.1
A_SIPEND        .equ 0x0010  ; Interrupt Pending register (6.13.2.1)
A_SIMASK        .equ 0x0014  ; Mask (6.13.2.2)
A_SIEM          .equ 0x0018  ; Level and edge for external IRQ lines
A_SIVEC         .equ 0x001C  ; Vector (level) when an interrupt occurs
A_PISCR         .equ 0x0240  ; (6.13.4.8)
A_PITC          .equ 0x0244  ; (6.13.4.10)
A_PITR          .equ 0x0248

.export Start_IHdlr

.text

InitInterrupt:
    stwu r1,-0x18(r1)    ; Save old SP into SP-0x18, SP=SP-0x18
    mfspr r0,LR           ; r0 = LR
    stw r30,0x10(r1)     ; [SP + 0x10] = r30
    stw r31,0x14(r1)     ; [SP + 0x14] = r31
    stw r0,0x1C(r1)      ; [SP + 0x1C] = r0
    li r0,0xA042          ; SET MSR !
andi. r0, r0, 0xFFFF ; keep only the 16 Low bit
mtmsr r0

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Use A_SIUBase as the base address for all of the modifications to the
; SIU registers (SIU = System Interface Unit)
lis r31, A_SIUBase
ori r31, r31, A_SIUOffset

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PITSCR - Periodic Interrupt Timer Status & Control Register
;
; PITSCR configuration
;
; Bits Description
; 0-7 PIRQ - Periodic Interrupt Request settings
;   Interrupt level to use for the PIT
; 8 PS - Periodic interrupt status
;   1 - Periodic interrupt has decremented past zero
;   0 - Periodic interrupt has not yet decremented past zero
;   (i.e. has it counted to zero yet?)
; 9-12 Reserved
; 13 PIE - Periodic Interrupt Enable
;   1 - Enable Interrupt, 0 - Disable Interrupt
; 14 PITF - Periodic Interrupt Freeze
;   1 - Will stop if freeze is asserted, 0 - Ignores freeze
; 15 PTE - Periodic timer enable
;   1 - Continue to decrement
;   0 - Stop counting (disable counter)
;
; Disable the periodic interrupt timer
li r0, 0x0480
; PIT irq Disable, and Stopped, set IRQ Level to 4
sth r0, A_PISCR(r31)

; Check to make sure the we got a valid value for the new PIT value
; (i.e. if it is zero, don't enable the PIT)
cmpi CR0, 0, r3, 0 ; If PITValue=0 Then disable PIT IRQ
beq IntI_00

; We got an OK value, write out the value to the PITSCR
sth r3, A_PITC(r31) ; Store PITValue (15 => 1,024 mS)

li r0, 0x3C85
; PIT irq enable, and running, set IRQ Level to 4
sth r0, A_PISCR(r31)

IntI_00:
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; SIMASK - SIU Interrupt Mask Register
;
; Select which interrupts to enable
; 1 -> Interrupt is allowed
0 -> Interrupt is masked (ignored)

Bits
0,1 IRQ0, Internal Interrupt 0
2,3 IRQ1, Internal Interrupt 1
4,5 IRQ2, Internal Interrupt 2
6,7 IRQ3, Internal Interrupt 3
8,9 IRQ4, Internal Interrupt 4
10,11 IRQ5, Internal Interrupt 5
12,13IRQ6, Internal Interrupt 6
14,15 IRQ7, Internal Interrupt 7
16-31 Reserved

`; Enable only the interrupt that we want, internal interrupt 4

lhz r0, A_SIMASK(r31) ; Get the original value
ori r0, r0, $0040 ; Enable interrupt level 4
; OR in the value because we want to preserve the
; current state of the interrupt mask

sth r0, A_SIMASK(r31)
; Write the changed setup back out to the interrupt controller

`; Restore the information from the stack

lwz r30,0x10(r1) ; restore register and LR
lwz r31,0x14(r1)
lwz r0,0x1C(r1)
mtspr LR,r0
addi r1,r1,0x18
blr

End_InitInterrupt:

`; Here is the interrupt handler, it will call the specific address located
; in InterruptTable and assigned at Link time.

Start_IHdldr:

stwu r1,-0x28(r1) ; Make a N word frame in the stack

stw r0,0x00C(r1) ; Save r0
mfspr r0,SRR0 ; SRR0 is a special register

stw r0,0x10(r1) ; Save SRR0
mfspr r0,SRR1

stw r0,0x14(r1) ; Save SRR1
mfspr r0,LR ; Get LR Link Register
stw r0,0x18(r1) ; Save it
mfcr r0 ; Get CR Condition Register

stw r0,0x1C(r1) ; Save it
stw r31,0x20(r1) ; Save it
stw r3,0x24(r1) ; R3 is ALWAYS used as result for a function!

;----------- Interrupt handler is here

li r31,0
addis r31,r31,A_SIUBase
ori r31,r31,0xC000

lbz r0,A_SIVEC(r31) ; r0 = Interrupt Vector
andi r0,r0,0x003C

li r31,InterruptTable@l
andi r31,r31,0xFFFF
addis r31,r31,InterruptTable@h
add r31,r31,r0
lwz r0,0(r31)
mtspr LR,r0 ; LR = r0
blrl ; Branch to R0

;----------- End of Interrupt handler

lwz r3,0x24(r1)
lwz r31,0x20(r1)
lwz r0,0x1C(r1) ; Get CR off stack
mtcrf 0xFF,r0 ; Restore CR Condition Register
lwz r0,0x18(r1) ; Get LR off stack
mtspr LR,r0 ; Restore LR Link Register
lwz r0,0x14(r1) ; Get SRR1 off stack
mtspr srr1,r0 ; Restore SRR1 Machine Status Save/Restore Reg 1
lwz r0,0x10(r1) ; Get SRR0 off stack
mtspr srr0,r0 ; Restore SRR0 Machine Status Save/Restore Reg 0
lwz r0,0x0C(r1) ; Restore r0
addi r1,r1,0x28 ; restore Stack: r1 = r1 + n

; Return from the interrupt

rfi

End_IHdlr:

.data

InterruptTable:
    .long InterruptHandlerIRQ0 ; 0
    .long InterruptHandler0 ; 1 Level 0x80 (0)
    .long InterruptHandlerIRQ1 ; 2
    .long InterruptHandler1 ; 3 Level 0x40 (1)
    .long InterruptHandlerIRQ2 ; 4
    .long InterruptHandlerIRQ2 ; 5 Level 0x20 (2)
.long InterruptHandlerIRQ3 ; 6
.long InterruptHandler3  ; 7 Level 0x10 (3)
.long InterruptHandlerIRQ4 ; 8
.long InterruptHandler4  ; 9 Level 0x08 (4)
.long InterruptHandlerIRQ5 ; 10
.long InterruptHandler5  ; 11 Level 0x04 (5)
.long InterruptHandlerIRQ6 ; 12
.long InterruptHandler6  ; 13 Level 0x02 (6)
.long InterruptHandlerIRQ7 ; 14
.long InterruptHandler7  ; 15 Level 0x01 (7)

; Making sure the interrupt handler gets included

.text

b IHdlr_Branch

.import Start_IHdlr
.export IHdlr_Branch

.function "IHdlr_Branch", IHdlr_Branch, End_IHdlr_Branch-IHdlr_Branch

.section .abs.FFF00500
.org 0

; .text

IHdlr_Branch:
    b    Start_IHdlr
End_IHdlr_Branch: