Memory and Data Structures

Arrays, Stacks, Queues

(Ch 10 & 16)
Midterm Overview

- May 11th, 2017 1:30PM-3:05PM
  - Scheduled for whole class
- No books, notes or calculators
- Show your work.
  - No credit if we don’t know how you did something
  - Lots of partial credit
- Practice midterm on canvas
- Seating chart
Memory

- This is the “RAM” in a system
- We have seen labels and addresses point to pieces of memory storing:
  - words
  - bytes
  - strings
  - numbers
- Memory is just a collection of bits
- We could use it to represent integers
- Or as an arbitrary set of bits
Treat memory as a giant array

- Compiler or programmer decides what use to make of it.
- The element numbering starts at 0
- The element number is an address
- In “C” to allocate some memory:

```c
char m[size_of_array];
```
Storage of Data

- LC-3 architecture is "word addressable" meaning that all addresses are "word" addresses.
- This means the smallest unit of memory we can allocate is 16-bits, a word.
- Use 'LD' (load) and 'ST' (store) to access this unit (or LDR & STR).
Example

mychar .BLKW 1
newline .FILL xA
...
...
LD R1, newline
GETC
ST R0, mychar
JSR Sub ; R2=R1-R0
BRz found_newline
...
found_newline ...
The data is placed in memory like this at start up (assuming data section starts at address 1). The “mychar” variable will change to the value of the character entered by the user once stored.
Pointers and Arrays

We've seen examples of both of these in our LC-3 programs, let's see how these work in "C"

**Pointer**
- Address of a variable in memory
- Allows us to indirectly access variables
  - in other words, we can talk about its *address* rather than its *value*

**Array**
- A list of values arranged sequentially in memory
- Example: a list of telephone numbers
- Expression `a[4]` refers to the 5th element of the array `a`
Arrays

Array implementation is very important

- Most assembly languages have only basic concept of arrays (BLKW)
- From an array, any other data structure we might want can be built
Properties of arrays:
- Each element is the same size
- Elements are stored contiguously
- First element at the smallest memory address

In assembly language we must
- Allocate correct amount of space for an array
- Map array addresses to memory addresses

\[ x [3008A] \]
LC-3 declarations of arrays within memory

To allocate a portion of memory (more than a single variable’s worth)

`variablename .BLKW numelements`

numelements is just that, numbering starts at 0 (as in "C")
Array of Integers

Calculating the address of an array element

```c
int myarray[7] /* C */
```

- If base address of “myarray” is 25

![Array diagram]

- Which is base address + distance from the first element
How do you get the address of myarray?

- Use the “load effective address” instruction, “LEA”
- Keep clear the difference between an address and the contents of an address.
To get address of `myarray[4]` in LC-3, write the code...

```
LEA  R0, myarray
ADD  R1, R0, 4
```

Now, if we wanted to increment element number 5 by 1...

```
LDR  R4, R1, 0
ADD  R4, R4, 1
STR  R4, R1, 0
```
Address vs. Value

Sometimes we want to deal with the address of a memory location, rather than the value it contains.

Recall example from Chapter 6: adding a column of numbers.
- R2 contains address of first location.
- Read value, add to sum, and increment R2 until all numbers have been processed.

R2 is a pointer -- it contains the address of data we’re interested in.
2-Dimensional Arrays

2-Dimensional arrays are more complicated in assembly

- Memory is a 1-D array
- Must map 2-D array to 1-D array
- Arrays have rows and columns
  - \( r \times c \) array
  - \( r = \text{rows} \)
  - \( c = \text{columns} \)
Two sensible ways to map 2-D to 1-D

Row major form: (rows are all together)

Column major form: (columns are all together)
3. 1 3.2 + 1 = 7

How do you calculate addresses in a 2-D array?

• Row Major:

Address \((r_i, c_i) =\) Base Address + \(((r_i \times \text{Number of Cols}) + c_i) \times \text{Element size})\)

• Column Major:

Address \((r_i, c_i) =\) Base Address + \(((c_i \times \text{Number of Rows}) + r_i) \times \text{Element size})\)
Summary of 2D arrays

- Row/Column major (storage order)
- Base address
- Size of elements
- Dimensions of the array

How about 3-D arrays?
Bounds Checking

- Many HLL’s have bounds checking (not C!!!)
- Assembly languages have no implied bounds checking
- Your program is in total control of memory
- With a 5 x 3 array, what does the following address?

```
array   .BLKW 15
LEA     R1, array
ADD     R1, R1, 15
LDR     R0, R1, 0
```

- Bounds checking is often a good idea!!
- Most C development environments include optional bounds checking.
HeartBleed

soo > "A"
Stacks

A LIFO (last-in first-out) storage structure.

– The first thing you put in is the last thing you take out.
– The last thing you put in is the first thing you take out.

This means of access is what defines a stack, not the specific implementation.

Two main operations:

**PUSH:** add an item to the stack

**POP:** remove an item from the stack
A Physical Stack

Coin rest in the arm of an automobile

Initial State

After One Push

After Three More Pushes

After One Pop

First quarter out is the last quarter in.
A Hardware Implementation

Data items move between registers

Initial State

Empty: Yes

After One Push

Empty: No

After Three More Pushes

Empty: No

After Two Pops
A Software Implementation

**Stack Overflow**

Data items don't move in memory, just our idea about where the TOP of the stack is.

Initial State | After One Push | After Three More Pushes | After Two Pops
--- | --- | --- | ---
\( \text{x4000} \) | \( \text{x3FFF} \) | \( \text{x3FFC} \) | \( \text{x3FFE} \)

By convention, R6 holds the Top of Stack (TOS) pointer.
Basic Push and Pop Code

For our implementation, stack grows downward (when item added, TOS moves closer to 0)

**Push**

- ADD R6, R6, #-1; decrement stack ptr
- STR R0, R6, #0; store data (R0)

**Pop**

- LDR R0, R6, #0; load data from TOS
- ADD R6, R6, #1; increment stack ptr
Pop with Underflow Detection

If we try to pop too many items off the stack, an underflow condition occurs.

- Check for underflow by checking TOS before removing data.
- Return status code in R5 (0 for success, 1 for underflow)

```
POP    LD    R1, EMPTY ; EMPTY = -x4000
ADD    R2, R6, R1 ; Compare stack pointer
BRz    FAIL ; to x4000 to see if empty
LDR    R0, R6, #0
ADD    R6, R6, #1
AND    R5, R5, #0 ; SUCCESS: R5 = 0
RET
FAIL   AND    R5, R5, #0 ; FAIL: R5 = 1
ADD    R5, R5, #1
RET
EMPTY  .FILL xC000 ; 2SC rep of –x4000
```
Push with Overflow Detection

If we try to push too many items onto the stack, an overflow condition occurs. This example assumes stack has room for 5 items.

- Check for overflow by checking TOS before adding data.
- Return status code in R5 (0 for success, 1 for overflow)

```
PUSH   LD R1, MAX ; MAX = -x3FFB
ADD    R2, R6, R1 ; Compare stack pointer
BRz    FAIL  ; top address to see if full
  ADD R6, R6, #-1
  STR R0, R6, #0
  AND R5, R5, #0 ; SUCCESS: R5 = 0
  RET
FAIL   AND R5, R5, #0 ; FAIL: R5 = 1
  ADD R5, R5, #1
  RET
MAX    .FILL xC005 ; 2SC of ~x3FFB
```
Stack Example

- Printing out a positive integer, character by character
- Push LSB to MSB
- Pop MSB to LSB (LIFO)

integer = 1024

if integer == 0 then
  push '0'
else
  while integer != 0
    digit ← integer mod base
    char ← digit + 48
    push char onto stack
    integer ← integer div base
  end while
end if

while stack is not empty
  pop char
  print char
RPN

Arithmetic Using a Stack

Instead of registers, some ISA's use a stack for source and destination operations: a zero-address machine.

- Example:
  ADD instruction pops two numbers from the stack, adds them, and pushes the result to the stack.

Evaluating $(A+B) \cdot (C+D)$ using a stack:

1. push $A$
2. push $B$
3. ADD
4. push $C$
5. push $D$
6. ADD
7. MULTIPLY
8. pop result

Why use a stack?
- Limited registers.
- Convenient calling convention for subroutines.
- Algorithm naturally expressed using LIFO data structure.
Example: OpAdd

POP two values, ADD, then PUSH result.
Example: OpAdd

OpAdd  JSR POP  ; Get first operand.
ADD R5,R5,#0  ; Check for POP success.
BRp Exit  ; If error, bail.
ADD R1,R0,#0  ; Make room for second.
JSR POP  ; Get second operand.
ADD R5,R5,#0  ; Check for POP success.
BRp Restore1  ; If err, restore & bail.
ADD R0,R0,R1  ; Compute sum.
JSR RangeCheck  ; Check size.
BRp Restore2  ; If err, restore & bail.
JSR PUSH  ; Push sum onto stack.
RET

Restore2  ADD R6,R6,#-1  ; Decr stack ptr (undo POP)
Restore1  ADD R6,R6,#-1  ; Decr stack ptr
RET
Queues

A queue is a FIFO (First In, First Out).
- The classic analogy of a queue is a line.
  - Person gets on the end of the line (the **Tail**),
  - Waits,
  - Gets off at the front of the line (the **Head**).
- Getting into the queue is an operation called **enqueue**
- Taking something off the queue is an operation called **dequeue**.
- It takes 2 pointers to keep track of the data structure,
  - Head (let’s use R5)
  - Tail always points to empty element (R6)
Initial state:

After 1 enqueue operation:

After another enqueue operation:
After a dequeue operation:

\[ \text{X Y} \]

Head (R5) \[ \text{Tail (R6)} \]

Like stacks, when an item is removed from the data structure, it is physically still present, but correct use of the structure cannot access it.
Implementation of a queue

Storage:

queue .BLKW infinity ; assume infinite for now
LEA R5, queue ; head
LEA R6, queue ; tail

Enqueue (item):

STR R0, R6, #0 ; R0 has data to store
ADD R6, R6, #1

Dequeue (item):

JSR SUB ; R0 = R5-R6
BRz queue_empty
LDR R1, R5, #0 ; put data in R1
ADD R5, R5, #1
Circular Queues

- To avoid infinite array, wrap around from end to beginning.
- Head == Tail means empty
- Head points to first item (for next dequeue)
- Tail points to empty location (for next enqueue)

Example of an 8 element circular queue
After "enqueue’ing" one element

After "enqueue’ing" another element
After “dequeuing” an element
Storage and initialization:

```
queue          .BLKW queue_size
queue_end      .BLKW 1
LEA           R5, queue ; head
LEA           R6, queue ; tail
```

Enqueue (item)

```
STR           R0, R6, #0 ; data to enqueue is in R0
ADD           R6, R6, #1
LEA           R1, queue_end
JSR           SUB        ; R1 = R1 - R6
BRp           continue1
LEA           R6, queue  ; wrap around
continue1
```
Dequeue (item):

JSR               SUB            ; R1 = R5 - R6
BRz               queue_empty
LDR               R0, R5, #0
ADD               R5, R5, #1
LEA               R1, queue_end
JSR               SUB            ; R1 = R5 - R1
BRn               continue2
LEA               R5, queue      ; wrap around
continue2
Summary of data structures

• All data structures are based on the simple array.
• 2D Arrays, Stacks, Queues.
• It is all about the implementation.
• Bounds checking is important.
• If not documented can become confusing.
Microcontrollers and Embedded Systems

Taking over the world
What is a microcontroller?

- A microprocessor
- Usually not cutting edge
- Dependable
  - All major bugs well known
- Predictable
  - Critical for real-time processing
- On-chip peripherals and memory
- Parallel and serial digital I/O
- Analog I/O
- Counters and timers
- Internal ROM and/or EPROM

Referred to as a “System On a Chip” (SoC)
What are microcontrollers used in?

- EVERYTHING

Some products that you might know:

- NASA’s Sojourner Rover – 8-bit Intel 80C85
- Palm Vx handheld – 32-bit Motorola Dragonball EZ
- Sonicare toothbrush – 8-bit Zilog Z8
- The Vendo V-MAX 720 Soda Machine – Motorola HC11
- Miele dishwasher – 8-bit Motorola 68HC05
- Hunter 44550 Programmable Thermostat – (4-bit cpu)

Pic32 - $9

30¢ millions 2¢
10 million
spec sheet
is out of spec
What word size are they? 32-bit is starting to dominate.
Microcontroller unit sales are much higher than microprocessors.
... and are MUCH, MUCH cheaper.
Dynamics of TOTAL CPU unit sales is changing
So what languages are they being programmed in?

<table>
<thead>
<tr>
<th>Language</th>
<th>‘98-’99</th>
<th>‘99-’00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td>~ 21%</td>
<td>~ 10%</td>
</tr>
<tr>
<td>C</td>
<td>~ 69%</td>
<td>~ 80%</td>
</tr>
<tr>
<td>C++</td>
<td>~ 5%</td>
<td>~ 6%</td>
</tr>
<tr>
<td>Java</td>
<td>~ 1%</td>
<td>~ 2%</td>
</tr>
<tr>
<td>Other</td>
<td>~ 3%</td>
<td>~ 2%</td>
</tr>
</tbody>
</table>

Global programming language usage

Most Popular Coding Languages of 2014

Python 30.3%
Java 22.2%
Ruby 10.6%
JavaScript 5.2%
C# 5%
C 4.1%
PHP 3.3%

Less popular languages:
- C++ 13%
- TCL .03%
- Lua .04%
- Bash .1%
- Clojure .2%
- Objective C .4%
- Scala 1%
$100 million  $1  2

Pintt Fingernail
10 million
Summary

- Microcontrollers/SoC are now the most common devices programmed
- The consist of a CPU plus a lot of other stuff
  - Built in IO
  - Built in audio/video
  - Built in sensors
  - etc
PIC32MX 320F128H OR 340F512H

- 80MHz processor
- 32-bit words (vs 16-bit LC3 words)
- 32 32-bit registers (vs 8 LC3 registers)
- Single cycle multiply (vs none on LC3)
- Multi-cycle divide (vs none on LC3)
- Memory
  - 128KB Flash
  - 16KB SRAM
- Many peripherals (Timers, I2C, UART, etc.)
  - 16-channel 10-bit Analog to Digital Converter
  - A real “System on a Chip” (SOC)
MIPS4k ISA

- The processor is a PIC32, but the ISA is MIPS4k. What does this mean?
  - PIC32 is an implementation of the MIPS 4000 ISA by Microchip Technology, Inc.
  - PIC32MX320F128.pdf talks about the specific implementation.
    - There are many versions of this chip with different memory configurations, peripherals, pin outs, etc.
    - Our chip: PIC32MX320F128 or PIC32MX340F512H

- Chapter 27 of the PDF is about the ISA
  - About 117 instructions – no need to memorize them all!!
  - Many you won’t use, but...
  - We will use a subset similar to LC3.
  - MIPS_Vol2.pdf also contains the ISA details
Hello World

#include <WProgram.h>

/* define all global symbols here */
.global main

.text
.set noreorder

.ent main
main:

/* your code goes underneath this */
LA $a0, HelloWorld
JAL puts
NOP
.end main

.data
HelloWorld: .asciiz "Hello World \n"

C compiler
assembly
FIGURE 1-1: BLOCK DIAGRAM

Note 1: Some features are not available on all device variants.
Note 2: BOR functionality is provided when the on-board voltage regulator is enabled.
# Register File(s)

## Table 1: Register Conventions

<table>
<thead>
<tr>
<th>CPU Register</th>
<th>Symbolic Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>zero</td>
<td>Always 0 (note 1)</td>
</tr>
<tr>
<td>r1</td>
<td>at</td>
<td>Assembler Temporary</td>
</tr>
<tr>
<td>r2 - r3</td>
<td>v0-v1</td>
<td>Function Return Values</td>
</tr>
<tr>
<td>r4 - r7</td>
<td>a0-a3</td>
<td>Function Arguments</td>
</tr>
<tr>
<td>r8 - r15</td>
<td>t0-t7</td>
<td>Temporary – Caller does not need to preserve contents</td>
</tr>
<tr>
<td>r16 - r23</td>
<td>s0-s7</td>
<td>Saved Temporary – Caller must preserve contents</td>
</tr>
<tr>
<td>r24 - r25</td>
<td>t8 - t9</td>
<td>Temporary – Caller does not need to preserve contents</td>
</tr>
<tr>
<td>r26 - r27</td>
<td>k0 - k1</td>
<td>Kernel temporary – Used for interrupt and exception handling</td>
</tr>
<tr>
<td>r28</td>
<td>gp</td>
<td>Global Pointer – Used for fast-access common data</td>
</tr>
<tr>
<td>r29</td>
<td>sp</td>
<td>Stack Pointer – Software stack</td>
</tr>
<tr>
<td>r30</td>
<td>s8 or fp</td>
<td>Saved Temporary – Caller must preserve contents OR Frame Pointer – Pointer to procedure frame on stack</td>
</tr>
<tr>
<td>r31</td>
<td>ra</td>
<td>Return Address (note 1)</td>
</tr>
</tbody>
</table>

Note 1: Hardware enforced, not just convention
Register File (2)

- Temporary can be used $t0...$t9
  - Procedures can destroy these
- Saved can be used $s0..$s7, but are caller save
  - Procedures must save/restore these
- $zero (or $0) is always 0
- $a0..a3 Are passed to functions as parameters
- $v0..v1 Are returned from functions
Stack

- Similar to LC-3... more later.
  - Stack Pointer (SP)
  - Frame Pointer (FP)
MIPS ISA

- MIPS_Vol2.pdf
- Chapter 27 of PIC32 manual
**Instruction Types**

**I-Type (Immediate)**
- 31 26 25 21 20 16 15 0
- op rs rt immediate

**J-Type (Jump)**
- 31 26 25 0
- op target

**R-Type (Register)**
- 31 26 25 21 20 16 15 11 10 6 5 0
- op rs rt rd sa funct

- **op**: 6-bit operation code
- **rs**: 5-bit source register specifier
- **rt**: 5-bit target (source/destination) register or branch condition
- **immediate**: 16-bit immediate value, branch displacement or address displacement
- **target**: 26-bit jump target address
- **rd**: 5-bit destination register specifier
- **sa**: 5-bit shift amount
- **funct**: 6-bit function field

*Figure 10-1 Instruction Formats*
Load Instructions

- 16-bits
- Byte/Half/Word accesses
- Only allow base + 16-bit immediate offset format
- Load data is not available to next instruction
  - "Load delay slot"
  - But processor should insert a wait cycle to get it (affects performance)
- DO NOT use unaligned load/store (LWL, LWR, SWL, SWR)
  - Addresses are bytes, so lower two bits should always be 00
- Recommend not using atomic load/store (LL, SC)
\[ t_0 \leq m \]
\[ t_1 = t_0 + t_2 \]
\[ t_3 = 56 + 57 \]

Cache
Load/Stores

- Typical load/stores for bytes, half word, word

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>Load Byte</td>
<td>( \text{Rt} = \text{(byte)} \text{Mem[Rs+offset]} )</td>
</tr>
<tr>
<td>LBU</td>
<td>Unsigned Load Byte</td>
<td>( \text{Rt} = \text{(ubyte)} \text{Mem[Rs+offset]} )</td>
</tr>
<tr>
<td>LH</td>
<td>Load Halfword</td>
<td>( \text{Rt} = \text{(half)} \text{Mem[Rs+offset]} )</td>
</tr>
<tr>
<td>LHU</td>
<td>Unsigned Load Halfword</td>
<td>( \text{Rt} = \text{(uhalf)} \text{Mem[Rs+offset]} )</td>
</tr>
<tr>
<td>LW</td>
<td>Load Word</td>
<td>( \text{Rt} = \text{Mem[Rs+offset]} )</td>
</tr>
<tr>
<td>LWPC</td>
<td>Load Word, PC relative</td>
<td>( \text{Rt} = \text{Mem[PC+offset]} )</td>
</tr>
<tr>
<td>SB</td>
<td>Store Byte</td>
<td>( \text{(byte)} \text{Mem[Rs+offset]} = \text{Rt} )</td>
</tr>
<tr>
<td>SH</td>
<td>Store Half</td>
<td>( \text{(half)} \text{Mem[Rs+offset]} = \text{Rt} )</td>
</tr>
<tr>
<td>SW</td>
<td>Store Word</td>
<td>( \text{Mem[Rs+offset]} = \text{Rt} )</td>
</tr>
</tbody>
</table>

- Other special load:
  - LUI, Load Upper Immediate, \( \text{Rt} = \text{immediate} \ll 16 \)
Slight Syntax Changes

• Register relative uses this syntax
  \[ \text{lw } t1, 4(t0) \]
  • Base register: \( t0 \)
  • Offset: 4 bytes
  • Destination register: \( t1 \)
  • Load the data from the address stored in \( t0+4 \) (\( \text{Mem}[t0+4] \)) into \( t1 \)

• No indirect addressing!
• Labels
  \[ \text{.text} \]
  lw \( $ra, \text{myStr} \)
  \[ \text{.data} \]
  myStr: \( \text{.asciiz "Hello, world!\n"} \)
Data Declarations

- **.text vs .data segment**
  - Loader can put these in different parts of memory
    - For example, text->flash and data->SRAM
    - Perhaps read-only permissions on .text segment

- **Declarations**

```assembly
var1: .word 3             # create a single integer variable with initial value 3
array1: .byte 'a','b'     # create a 2-element character array with elements initialized to a and b
array2: .space 40         # allocate 40 consecutive bytes, with storage uninitialized

string1: .asciiz "Hello!\n" # string variable with end null
string2: .asciiz "Hello!\n" # string variable with NO end null
```
### Arithmetic/Logical

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Integer Add</td>
<td>$R_d = R_s + R_t$</td>
</tr>
<tr>
<td>ADDI</td>
<td>Integer Add Immediate</td>
<td>$R_t = R_s + \text{Immed}$</td>
</tr>
<tr>
<td>ADDIU</td>
<td>Unsigned Integer Add Immediate</td>
<td>$R_t = R_s +_U \text{Immed}$</td>
</tr>
<tr>
<td>ADDU</td>
<td>Unsigned Integer Add</td>
<td>$R_d = R_s +_U R_t$</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
<td>$R_d = R_s &amp; R_t$</td>
</tr>
<tr>
<td>ANDI</td>
<td>Logical AND Immediate</td>
<td>$R_t = R_s &amp; \text{Immed}$</td>
</tr>
<tr>
<td>NOR</td>
<td>Logical NOR</td>
<td>$R_d = \lnot (R_s \lor R_t)$</td>
</tr>
<tr>
<td>OR</td>
<td>Logical OR</td>
<td>$R_d = R_s \lor R_t$</td>
</tr>
<tr>
<td>ORI</td>
<td>Logical OR Immediate</td>
<td>$R_t = R_s \lor \text{Immed}$</td>
</tr>
<tr>
<td>SUB</td>
<td>Integer Subtract</td>
<td>$R_t = (\text{int})R_s - (\text{int})R_d$</td>
</tr>
<tr>
<td>SUBU</td>
<td>Unsigned Subtract</td>
<td>$R_t = (\text{uns})R_s - (\text{uns})R_d$</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive OR</td>
<td>$R_d = R_s ^ R_t$</td>
</tr>
<tr>
<td>XORI</td>
<td>Exclusive OR Immediate</td>
<td>$R_t = R_s ^ (\text{uns})\text{Immed}$</td>
</tr>
</tbody>
</table>
Computational Instructions

- R-type
- Arithmetic, logical, shift...

### Add Word

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>ADD</td>
<td>100000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  
ADD rd, rs, rt

**MIPS32 (MIPS I)**

**Purpose:**
To add 32-bit integers. If an overflow occurs, then trap.

**Description:**  
rd ← rs + rt

The 32-bit word value in GPR rt is added to the 32-bit value in GPR rs to produce a 32-bit result.

- If the addition results in 32-bit 2’s complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is placed into GPR rd.
Computational Instructions 2

- **I-type**
- **Arithmetic, logical, shift, ...**

### Add Immediate Word

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** ADDI rt, rs, immediate

**Purpose:**

To add a constant to a 32-bit integer. If overflow occurs, then trap.

**Description:** rt ← rs + immediate

The 16-bit signed immediate is added to the 32-bit value in GPR rs to produce a 32-bit result.

- If the addition results in 32-bit 2’s complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is placed into GPR rt.
Signed vs Unsigned Instructions

- Example: ADDIU vs ADDI
  - No sign extension of immediates
  - No overflow conditions (modulo arithmetic)
- Example: ANDI
  - No sign extension
Pseudo-ops

• Our assembler does a better job of "guessing" types
  – Will often correct errors with pseudo-ops using $at register
  • Don’t use the $at register!
  – Example:  \texttt{add \$t0,\$a0,0x12345678}

BECOMES

\texttt{lui \$at,0x1234}
\texttt{ori \$at,\$at,0x5678}
\texttt{add \$t0,\$a0,\$at}
Shift/Rotate

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROTR</td>
<td>Rotate Word Right</td>
<td>$Rd = Rt[sa−1:0] \mid Rt[31:sa]$</td>
</tr>
<tr>
<td>ROTRV</td>
<td>Rotate Word Right Variable</td>
<td>$Rd = Rt[Rs−1:0] \mid Rt[31:Rs]$</td>
</tr>
<tr>
<td>SLL</td>
<td>Shift Left Logical</td>
<td>$Rd = Rt &lt;&lt; sa$</td>
</tr>
<tr>
<td>SLLV</td>
<td>Shift Left Logical Variable</td>
<td>$Rd = Rt &lt;&lt; Rs[4:0]$</td>
</tr>
<tr>
<td>SRA</td>
<td>Shift Right Arithmetic</td>
<td>$Rd = (int)Rt &gt;&gt; sa$</td>
</tr>
<tr>
<td>SRAV</td>
<td>Shift Right Arithmetic Variable</td>
<td>$Rd = (int)Rt &gt;&gt; Rs[4:0]$</td>
</tr>
<tr>
<td>SRL</td>
<td>Shift Right Logical</td>
<td>$Rd = (uns)Rt &gt;&gt; sa$</td>
</tr>
<tr>
<td>SRLV</td>
<td>Shift Right Logical Variable</td>
<td>$Rd = (uns)Rt &gt;&gt; Rs[4:0]$</td>
</tr>
</tbody>
</table>
# Conditional Set Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
</table>
| SLT         | Set on Less Than                         | if (int)Rs < (int)Rt  
                                     | Rd = 1                                      |
|             |                                          | else                                        |
|             |                                          | Rd = 0                                      |
| SLTI        | Set on Less Than Immediate               | if (int)Rs < (int)Immed  
                                     | Rt = 1                                      |
|             |                                          | else                                        |
|             |                                          | Rt = 0                                      |
| SLTIU       | Set on Less Than Immediate Unsigned      | if (uns)Rs < (uns)Immed  
                                     | Rt = 1                                      |
|             |                                          | else                                        |
|             |                                          | Rt = 0                                      |
| SLTU        | Set on Less Than Unsigned                | if (uns)Rs < (uns)Immed  
                                     | Rd = 1                                      |
|             |                                          | else                                        |
|             |                                          | Rd = 0                                      |
Branches

- Perform comparison/check AND branch!

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Unconditional Branch (Assembler idiom for: BEQ r0, r0, offset)</td>
<td>PC += (int)offset</td>
</tr>
<tr>
<td>BEQ</td>
<td>Branch On Equal</td>
<td>if Rs == Rt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC += (int)offset</td>
</tr>
<tr>
<td>BGEZ</td>
<td>Branch on Greater Than or Equal To Zero</td>
<td>if !Rs[31]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC += (int)offset</td>
</tr>
<tr>
<td>BGTZ</td>
<td>Branch on Greater Than Zero</td>
<td>if !Rs[31] &amp; Rs != 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC += (int)offset</td>
</tr>
<tr>
<td>BLEZ</td>
<td>Branch on Less Than or Equal to Zero</td>
<td>if Rs[31]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC += (int)offset</td>
</tr>
<tr>
<td>BLTZ</td>
<td>Branch on Less Than Zero</td>
<td>if Rs[31]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC += (int)offset</td>
</tr>
<tr>
<td>BNE</td>
<td>Branch on Not Equal</td>
<td>if Rs != Rt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC += (int)offset</td>
</tr>
</tbody>
</table>
Jumps

- jal foo to call subroutine
- jr $ra to return

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>Unconditional Jump</td>
<td>PC = PC[31:28]</td>
</tr>
<tr>
<td>JAL</td>
<td>Jump and Link</td>
<td>GPR[31] = PC + 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC = PC[31:28]</td>
</tr>
<tr>
<td>JALR</td>
<td>Jump and Link Register</td>
<td>Rd = PC + 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC = Rs</td>
</tr>
<tr>
<td>JR</td>
<td>Jump Register</td>
<td>PC = Rs</td>
</tr>
</tbody>
</table>
Control Instructions

- BEWARE: Jump and branch also have a “delay slot”
  - First instruction after is ALWAYS executed!
  - Put a NOP if you are unsure, but this loses performance.
  - NOP is “no operation”

```
addiu $t4,$t4,-1
bgtz $t4, targetlabel
addiu $t5,$t5,1
```

```
addiu $t4,$t4,-1
bgtz $t4, targetlabel
nop
addiu $t5,$t5,1
```
Delay Slots Nuances

• Load
  – If the instruction after a load uses the result, a nop will be inserted by hardware.
  – Can effect performance by increasing number of instructions
  – Can reorder instructions to avoid this dependency

• Branch
  – Instruction after a branch is ALWAYS executed whether branch is taken or not
  – Can insert a nop to fill this
  – Can insert another instruction to make this slot useful
Condition Codes?

- Does the condition codes of the branch delay slot affect the branch?
  - MIPS does not have condition codes!
  - The branch itself does the comparison.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Unconditional Branch (Assembler idiom for: BEQ r0, r0, offset)</td>
<td>PC += (int)offset</td>
</tr>
<tr>
<td>BEQ</td>
<td>Branch On Equal</td>
<td>if Rs == Rt \n PC += (int)offset</td>
</tr>
<tr>
<td>BGEZ</td>
<td>Branch on Greater Than or Equal To Zero</td>
<td>if !Rs[31] \n PC += (int)offset</td>
</tr>
<tr>
<td>BGTZ</td>
<td>Branch on Greater Than Zero</td>
<td>if !Rs[31] &amp;&amp; Rs != 0 \n PC += (int)offset</td>
</tr>
<tr>
<td>BLEZ</td>
<td>Branch on Less Than or Equal to Zero</td>
<td>if Rs[31]</td>
</tr>
<tr>
<td>BLTZ</td>
<td>Branch on Less Than Zero</td>
<td>if Rs[31] \n PC += (int)offset</td>
</tr>
<tr>
<td>BNE</td>
<td>Branch on Not Equal</td>
<td>if Rs != Rt \n PC += (int)offset</td>
</tr>
</tbody>
</table>
Multiply (and accumulate)

- Takes one cycle to perform
- Result of a 32x32-bit multiply is 64-bits...
  - Two extra 32-bit registers: HI and LO
  - MFHI and MFLO opcodes to retrieve results
    - MFHI $t0 or MFLO $t1
  - MTHI and MTLO opcodes to load HI/LO register
- MULT $t1,$t0 or MULTU $t1,$t0
  - (HI,LO) = (int)$t1*(int)$t0
  - (HI,LO) = (uns)$t1*(uns)$t0
- MUL $t2, $t1,$t0
  - $t2=$t1*$t0
Divide

- Uses HI/LO registers as well
- Takes many cycles to perform
- DIV
  - \( LO = (\text{int})Rs / (\text{int})Rt \)
  - \( HI = (\text{int})Rs \% (\text{int})Rt \)
- DIFU
  - \( LO = (\text{uns})Rs / (\text{uns})Rt \)
  - \( HI = (\text{uns})Rs \% (\text{uns})Rt \)
Useful PseudoOps

• move $v0,$zero
  – addu $v0,$zero,$zero

• li $t0,0x01234567
  – lui $t0,0x0123
  – addiu $t0,$t0,0x4567

• la $t0,main
  – Similar to above but loads a label address into a register
Examples

- To initialize a register
  - `move $t0,$zero`

- To load immediate into a register
  - `lui $t0, 0xBF88`
  - `addiu $t0,$t0, 0x6140`

- Lots more examples
  - [http://www.johnloomis.org/microchip/pic32/resources.html](http://www.johnloomis.org/microchip/pic32/resources.html)