PIC32

PIC32 Overview
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)

40 million I/O per second
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](attachment: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](attachment: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"

Maxwell James Dunne
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)

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

J-Type (Jump)

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>target</td>
<td></td>
</tr>
</tbody>
</table>

R-Type (Register)

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>6 5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>funct</td>
<td></td>
</tr>
</tbody>
</table>

- 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
- 8 16 32
- 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)
\[ t_0 < m \]
\[ t_1 = t_0 + t_2 \]
\[ t_3 = 56 + 57 \]

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

- Other special load:
  - LUI, Load Upper Immediate, $Rt = \text{immediate} \ll 16$
<table>
<thead>
<tr>
<th>LUI</th>
<th>32</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Slight Syntax Changes

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

- No indirect addressing!

- Labels
  \[ \text{.text} \]
  \[ \text{lw} \quad \text{ra, myStr} \]
  \[ \text{.data} \]
  \[ \text{myStr: .asciiz "Hello, world!\n"} \]
LDR R1, R0, 3

R0 + 3 → Mem

Mem[R0 + 3]

LW$t1, 3($t0)
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**

  ```
  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: .ascii "Hello!\n"   # string variable with NO end null
  ```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Integer Add</td>
<td>(Rd = Rs + Rt)</td>
</tr>
<tr>
<td>ADDI</td>
<td>Integer Add Immediate</td>
<td>(Rt = Rs + \text{Immed})</td>
</tr>
<tr>
<td>ADDIU</td>
<td>Unsigned Integer Add Immediate</td>
<td>(Rt = Rs +_U \text{Immed})</td>
</tr>
<tr>
<td>ADDU</td>
<td>Unsigned Integer Add</td>
<td>(Rd = Rs +_U Rt)</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
<td>(Rd = Rs &amp; Rt)</td>
</tr>
<tr>
<td>ANDI</td>
<td>Logical AND Immediate</td>
<td>(Rt = Rs &amp; \text{Immed})</td>
</tr>
<tr>
<td>NOR</td>
<td>Logical NOR</td>
<td>(Rd = \sim(Rs \mid Rt))</td>
</tr>
<tr>
<td>OR</td>
<td>Logical OR</td>
<td>(Rd = Rs \mid Rt)</td>
</tr>
<tr>
<td>ORI</td>
<td>Logical OR Immediate</td>
<td>(Rt = Rs \mid \text{Immed})</td>
</tr>
<tr>
<td>SUB</td>
<td>Integer Subtract</td>
<td>(Rt = (\text{int})Rs - (\text{int})Rd)</td>
</tr>
<tr>
<td>SUBU</td>
<td>Unsigned Subtract</td>
<td>(Rt = (\text{uns})Rs - (\text{uns})Rd)</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive OR</td>
<td>(Rd = Rs ^ Rt)</td>
</tr>
<tr>
<td>XORI</td>
<td>Exclusive OR Immediate</td>
<td>(Rt = Rs ^ (\text{uns})\text{Immed})</td>
</tr>
</tbody>
</table>
Computational Instructions

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

Add Word

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>ADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
<td>15</td>
<td>11</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>SPECIAL</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>
</tr>
</tbody>
</table>

Format: ADD rd, rs, rt

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

- l-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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDI</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td></td>
</tr>
</tbody>
</table>

Format: ADDI rt, rs, immediate

MIPS32 (MIPS I)

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

Description: \(rt \leftarrow rs + \text{immediate}\)

The 16-bit signed \text{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:
    \[
    \begin{align*}
    \text{add} & \quad \text{t0}, \quad \text{a0}, \quad 0x12345678 \\
    \text{lui} & \quad \text{at}, \quad 0x1234 \\
    \text{ori} & \quad \text{at}, \quad \text{at}, \quad 0x5678 \\
    \text{add} & \quad \text{t0}, \quad \text{a0}, \quad \text{at}
    \end{align*}
    \]
### 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>$Rd = Rt[sa-1:0]</td>
</tr>
<tr>
<td>ROTRV</td>
<td>Rotate Word Right Variable</td>
<td>$Rd = Rt[Rs-1:0]</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>
Rs \xleftarrow{3} \rightarrow 0
## 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>
</table>
| B           | Unconditional Branch  
(Assembler idiom for: BEQ r0, r0, offset) | PC += (int)offset |
| BEQ         | Branch On Equal | if Rs == Rt  
PC += (int)offset |
| BGEZ        | Branch on Greater Than or Equal To Zero | if !Rs[31]  
PC += (int)offset |
| BGTZ        | Branch on Greater Than Zero | if !Rs[31] && Rs != 0  
PC += (int)offset |
| BLEZ        | Branch on Less Than or Equal to Zero | if Rs[31] || Rs == 0  
PC += (int)offset |
| BLTZ        | Branch on Less Than Zero | if Rs[31]  
PC += (int)offset |
| BNE         | Branch on Not Equal | if Rs != Rt  
PC += (int)offset |
# 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"

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

```assembly
addiu $t5,$t5,1
```

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

```assembly
[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.
Add $t4,$t5,$s0
BGTZ $t4, FOO
Add $t1,$t2,3
ADD $t4,$t4,1
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&lt;br&gt;PC += (int)offset</td>
</tr>
<tr>
<td>BGEZ</td>
<td>Branch on Greater Than or Equal To Zero</td>
<td>if !Rs[31]&lt;br&gt;PC += (int)offset</td>
</tr>
<tr>
<td>BGTZ</td>
<td>Branch on Greater Than Zero</td>
<td>if !Rs[31] &amp;&amp; Rs != 0&lt;br&gt;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></td>
</tr>
<tr>
<td>BNE</td>
<td>Branch on Not Equal</td>
<td>if Rs != Rt&lt;br&gt;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**
  - \( \text{LO} = \text{(int)}Rs \div \text{(int)}Rt \)
  - \( \text{HI} = \text{(int)}Rs \% \text{(int)}Rt \)
- **DIFU**
  - \( \text{LO} = \text{(uns)}Rs \div \text{(uns)}Rt \)
  - \( \text{HI} = \text{(uns)}Rs \% \text{(uns)}Rt \)

24 cycles


```plaintext
var / 123
   6
var x Large number
```

```
Hi    | L 0
H1    |?
H1 + constant
```
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
PIC32

PIC32 Overview Pt II - Functions
Functions

• What are functions?
  – Same as subroutines, but can pass arguments and return values
  – function definition:
    • \texttt{squared(a)}
    \begin{verbatim}
    return(a*a)
    \end{verbatim}
  – function call:
    • \texttt{ret=squared(b)}

• Implemented as subroutine calls in assembly
  – Load variable (“b”) from memory into register argument
  – Jump to subroutine label “squared”
  – Put result in register for return
  – Return from subroutine call
  – Store result into variable (“ret”) in memory

Of course, the subroutine saves/restores registers!
Function Calls (Caller)

- Arguments are put in \texttt{a0..a3}
  - Less than or equal to 4 arguments
  - What if there’s more than 4?
- \texttt{ra} contains return address if “link” instruction is used (jal or jalr)
  - Example: \texttt{jal myfunction}
  - Example 2: \texttt{la $t0 myfunction}
    - \texttt{jalr $t0}
- Caller Duties
  - Caller must save $t0...$t9 if it wants to preserve them during function call
  - Caller must save \texttt{ra} if it is a subroutine itself!
Function Returns (Callee)

- Return values are put in $v0..$v1
  - Less than or equal to 2 return values
  - What happens if there are more?
- Return with jr $ra
- Callee duties
  - Must save/restore $s0..$s7 if it wants to USE them
Should we use local memory to save/restore?

• NO!
• Why not?
  – This makes the program bigger.
  – This means we can only call the function once.
• LC3 and MIPS both have function calling conventions.
  – This is on the previous slides. Follow them!
  – There are some more details, but you will not need them for our labs.
Push & Pop in LC3

- **Push**
  - Decrement TOS pointer (our stack is moving down)
  - then write data in R0 to new TOS

  \[
  \text{PUSH} \quad \text{ADD} \quad R6, \: R6, \: \# \: -1 \\
  \text{STR} \quad R0, \: R6, \: \# \: 0
  \]

- **Pop**
  - Read data at current TOS into R0
  - then increment TOS pointer

  \[
  \text{POP} \quad \text{LDR} \quad R0, \: R6, \: \# \: 0 \\
  \text{ADD} \quad R6, \: R6, \: \# \: 1
  \]
Push/Pop in MIPS

- Stack pointer ($sp$ or $r29$)
  
  push: addi $sp$, $sp$, -4  # Decrement stack pointer by 4
  sw $s3$, 0($sp)  # Save $s3$ to stack

  pop: lw $s3$, 0($sp)  # Copy from stack to $s3$
  addi $sp$, $sp$, 4  # Increment stack pointer by 4

- What if more than one word?
  
  \[
  32 \text{ byte addressable, } 8 \]
  
  \[
  32 \div 8 = 4
  \]
Multiple Push/Pop

- Can save instructions by doing multiple loads (or stores) to the stack and then incrementing the stack pointer one time
  - Fewer addi instructions

```
push:      addi  $sp, $sp,  -12 # Decrement stack pointer by 12
           sw    $s3, 0($sp)    # Save $s3 to stack
           sw    $s2, 4($sp)     # Save $s2 to stack
           sw    $s1, 8($sp)     # Save $s1 to stack

Pop:       lw     $s3, 0($sp) # Copy $s3 from stack
           lw     $s2, 4($sp)  # Copy $s2 from stack
           lw     $s1, 8($sp)  # Copy $s1 from stack
           addi  $sp, $sp, 12  # Increment stack pointer by 12
```
Stacks and Functions

- Stack frame is the part of the stack for a particular function
- More than 4 arguments will be put on the stack
After calling “foo”

- Foo() needs to save/restore registers
- $sp still points to top of stack
- $fp (or $r30) points to where top of stack WAS
  - Convenient if $sp grow/shrinks inside of our subroutine foo
  - $fp doesn’t move inside foo
Using $fp

- $fp points to a fixed location of return value and arguments
- Calling functions inside foo
  - Must save $fp to stack first
- To return from foo:
  - Copy all of your data from $sp
  - Then copy $fp into $sp rather than worry about the size of popped data
Important Note

- If main() calls foo(), then foo() calls bar()
  - Foo must save its arguments $a0..$a3 on the stack
  - Foo then puts new arguments on stack for bar
- “Leaf” functions don’t need to save arguments since they don’t call functions
Local Memory

• What if your function needs more memory?
  – Can also place temporary memory on the stack in a subroutine
  – This is better than statically allocating memory (e.g. .blkw in LC3)
What if I want global memory?

- Global memory is shared among several subroutines
  - It isn’t passed as an argument
  - It isn’t returned as a return value
  - Most programming classes suggest to avoid it because it can be modified ANYWHERE and lead to BUGS
- Global pointer ($gp$ or $r28$)
  - Start of .data segment
- Heap is memory managed by the operating system
  - Done with a system call
Static Global Data example

- $gp$ should point to beginning of .data segment
  
  - .data
  - myspacer: .space 4
  - myarray: .byte 1,2,3,4
  - .text
  - ld $s0, 0($gp) will load myspacer
  - lb $s1,4($gp) will load myarray[0]
  - lb $s1,5($gp) will load myarray[1]
  - ... (note BYTES)