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)}
      \texttt{return(a*a)}
  – Function call:
    • \texttt{ret=squared(b)}

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

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

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

  ```
  PUSH ADD R6, R6, # -1
  STR R0, R6, # 0
  ```

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

  ```
  POP LDR R0, R6, # 0
  ADD R6, R6, # 1
  ```

---

![Stack diagrams showing initial state, after one push, after three pushes, and after two pops.](image)
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?
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

```
SP →

foo

ret val

main

foo() stack frame

main() stack frame

args
```
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)
\text{LW} \text{ $51, 14596$ ($910$)
PIC32
PIC32 IO
What is an I/O Port?

I/O: Input / Output

Buttons and switches

H T A 3.3V 5V 12V
L F 0 0 0 0
Digital Ports (TriState)
Digital Ports (IO Section)
Digital Ports (IO Section)

Figure 12-1: Typical Port Structure Block Diagram

- RD ODCx
- Data Bus
- SYSCLK
- WR ODCx
- RD TRISx
- WR TRISx
- WR LATx
- WR PORTx
- RD LATx
- RD PORTx
- SLEEP
- SYSCLK

Dedicated Port Module

I/O Cell

I/O pin

Synchronization
Port Configuration

- **TRISG**
  - Configures whether input or output
  - Input by default

- **PORTG**
  - Read or write value from/to port

- **LATG**
  - Read or write latched value from/to port
Ethernet

Port D3

LAID30

Pin D3
PORTD Memory Mapped Regs

- Masks in MPLAB assembler
  - $1<<8$ is same as $0x0100$
  - $15<<8$ is same as $0x0f00$
  - $(\text{number})<<(\text{shift amount})$ with both being constants

---


<table>
<thead>
<tr>
<th>Virtual Address (BP8, #)</th>
<th>Register Name</th>
<th>Bit Range</th>
<th>31/15</th>
<th>30/14</th>
<th>29/13</th>
<th>28/12</th>
<th>27/11</th>
<th>26/10</th>
<th>25/9</th>
<th>24/8</th>
<th>23/7</th>
<th>22/6</th>
<th>21/5</th>
<th>20/4</th>
<th>19/3</th>
<th>18/2</th>
<th>17/1</th>
<th>16/0</th>
<th>All Resets</th>
</tr>
</thead>
<tbody>
<tr>
<td>60CD</td>
<td>TRISD</td>
<td>31:16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TRISD11</td>
<td>TRISD10</td>
<td>TRISD9</td>
<td>TRISD8</td>
<td>TRISD7</td>
<td>TRISD6</td>
<td>TRISD5</td>
<td>TRISD4</td>
<td>TRISD3</td>
<td>TRISD2</td>
<td>TRISD1</td>
<td>TRISD0</td>
<td>0000</td>
</tr>
<tr>
<td>60CC</td>
<td>PORTD</td>
<td>31:16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RD11</td>
<td>RD10</td>
<td>RD9</td>
<td>RD8</td>
<td>RD7</td>
<td>RD6</td>
<td>RD5</td>
<td>RD4</td>
<td>RD3</td>
<td>RD2</td>
<td>RD1</td>
<td>RD0</td>
<td>0000</td>
</tr>
<tr>
<td>60CE</td>
<td>LATD</td>
<td>31:16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LATD11</td>
<td>LATD10</td>
<td>LATD9</td>
<td>LATD8</td>
<td>LATD7</td>
<td>LATD6</td>
<td>LATD5</td>
<td>LATD4</td>
<td>LATD3</td>
<td>LATD2</td>
<td>LATD1</td>
<td>LATD0</td>
<td>0000</td>
</tr>
<tr>
<td>60CF</td>
<td>OCD</td>
<td>31:16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OCD11</td>
<td>OCD10</td>
<td>OCD9</td>
<td>OCD8</td>
<td>OCD7</td>
<td>OCD6</td>
<td>OCD5</td>
<td>OCD4</td>
<td>OCD3</td>
<td>OCD2</td>
<td>OCD1</td>
<td>OCD0</td>
<td>0000</td>
</tr>
</tbody>
</table>

Legend:  
$\times$ = unknown value on Reset, $\_\_\_\_$ = unimplemented, read as ‘0’. Reset values are shown in hexadecimal.

Note 1: All registers in this table have corresponding CLR, SET and INV registers at their virtual addresses, plus offsets of 0x4, 0x8 and 0xC, respectively. See Section 12.1.1 “CLR, SET and INV Registers” for more information.
Input Switches

- Port D
  - (SW1, SW2, SW3, SW4) -> (RD8, RD9, RD10, RD11)
  - TRISD
    - TRISDCLR
    - TRISDSET
    - TRISDINV
  - PORTD
    - PORTDCLR
    - PORTDSET
    - PORTDINV
  - LATD

DO be an output 000
Device Registers

- All device registers have 4 registers to configure them
  - Actual (offset +0): you can read or write the whole register
  - Atomic registers, why do we want these?
    - CLR (offset +4)
    - SET (offset +8)
    - INV (offset +C)

- Example:
  - TRISG can clear bit 0 by writing 0001 to TRISG + 4
  - Sw $a0, 4($t0)
4

3.14

1 week \rightarrow 2 weeks

Last lab
Cipher