Digital Logic Design
234262
Alex Bronstein

Lecture 9: Function calls, exceptions, interrupts
Abstraction Hierarchy

Application Software

Operating System *(threads, files, exceptions)*

Computer Architecture *(instruction set)*

Micro-Architecture *(execution pipeline)*

Logic *(adders, multipliers, FSMs)*

Digital Circuits *(gates)*

Analog Circuits *(amplifiers)*

Process and Devices *(MOS FET transistors)*

Physics *(electrons, holes, tunneling, band gap)*
Control instructions

- Unconditional jump
- Conditional branch
- Function call/return
- Exception (aka software interrupt)
- Hardware interrupt
Function calls

- Invoking a function changes the flow of a program twice:
  - **Calling** the function
  - **Returning** from the function

- Each time a function is called, PC is updated and CPU has to store return address

- Function accepts **arguments** and returns **return values**

```c
void main()
{
    ...
    a = fact(8);
    b = fact(3);
    c = fact(8-3);
    coef = a/b/c;
    ...
}

int fact(int n)
{
    int i, f=1;
    for (i=n; i>1; i--)
        f = f*i;
    return f;
}
```
Function calls in MIPS

- **Function call**: jump-and-link (jal) instruction:
  - jal (encoded as J-type) stores the PC of the **next instruction** in R31 and performs a jump
    \[
    \text{jal target} \iff \$31 \leftarrow \text{PC} + 4
    \]
    \[
    \text{j target}
    \]

- **Return from function**: jr $s
  - jr (encoded as R-type) sets PC to the value stored in the supplied Rs
  - The only MIPS instructions that can access PC
MIPS uses the following conventions:

- Up to four arguments are passed on registers $4$-$7$ (alias $a0$-$a3$)
- Up to two values can be returned on registers $2$-$3$ (alias $v0$-$v1$)

Note: assembly does not check types! A 32-bit value may be a signed/unsigned integer, single-precision floating point, or a pointer to an arbitrarily typed object.

Conventions are not enforced by hardware or assembler.

Compilers and assembly programmers must follow conventions so that functions written by different people interface with each other.

A “rogue” function may cause unpredictable results.
Nested calls

- Stack-like behavior: **last in-first out**
- Someone calls A
  - A calls B
    - B calls C
    - C returns to B
  - B returns to A
- A returns
- Calling C overwrites arguments of B in $a0-$a3 and return address in $ra
Register spilling/filling

- The number of CPU registers is limited and it is possible that several functions will conflict over the same registers.
- Important registers can be saved (spilling) before function executes and restored (filling) after function completes.
- **Who** is responsible for saving registers – the **caller** or the **callee**?
- **Where** are they saved?
Who saves the registers?

- The **caller** knows which registers are important to it and shall be saved across function call
- The **callee** knows exactly which register it will use and overwrite
- Good engineering practices is **black box**: the caller and callee do not know anything about each other’s implementation

- Possible conventions:
  - Caller saves them all
  - Callee saves them all
  - MIPS convention: split responsibilities between caller and callee
## MIPS register conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Alias</th>
<th>Conventional use</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2−$3</td>
<td>$v0−v1</td>
<td>values from expression evaluation &amp; function results</td>
</tr>
<tr>
<td>$4−$7</td>
<td>$a0−a3</td>
<td>arguments – first four arguments of a function</td>
</tr>
<tr>
<td>$8−$15</td>
<td>$t0−t7</td>
<td>temporaries</td>
</tr>
<tr>
<td>$16−$23</td>
<td>$s0−s7</td>
<td>saved values</td>
</tr>
<tr>
<td>$31</td>
<td>$ra</td>
<td>return address</td>
</tr>
</tbody>
</table>

* saved by caller when necessary (callee may modify these registers)
* saved by callee when necessary (caller may assume them unchanged by the callee)
* saved by callee that is also a caller
Where are the registers spilled?

- Part of the memory is used for **stack** (a last in first out data structure)
- Stack grows downwards in terms of addresses
- Top element of the stack is pointed by (by convention) by register $29$ (alias $sp$)
- No push/pop instructions built into the ISA
To push elements into the stack:
- Decrement $sp$ to make room for new data
- Store elements on the stack

To push registers $t1$, $t2$ onto the stack:

\[
\begin{align*}
\text{sub} & \quad sp, 8 \\
\text{sw} & \quad t1, 4(sp) \\
\text{sw} & \quad t2, 0(sp)
\end{align*}
\]

or alternatively:

\[
\begin{align*}
\text{sw} & \quad t1, -4(sp) \\
\text{sw} & \quad t2, -8(sp) \\
\text{sub} & \quad sp, 8
\end{align*}
\]
Pop

- Elements can be **accessed** relative to $sp
- To retrieve the value of $t1:
  
  ```
  lw   $s0,4($sp)
  ```
- To **pop** elements from the stack increment $sp
  
  ```
  add  $sp,8
  ```
- Data past stack pointer still remain in memory but are considered invalid
- Arguments can be passed to function on stack
Example of function call

$sp \rightarrow \text{...}

A: \text{jal} B
   \text{...}

B: \text{jal} C
   \text{jr} $ra

B2: \text{...}
   \text{jr} $ra

C: \text{...}
Example of function call

```
A: ...
jal B
...

B: ...
jal C

B2: ...
jr $ra

C: ...
jr $ra
```

.saved
$v*, $a*,
$t*, $ra

$sp

B args

PC

$ra

saved
$sp

$ra

$ra

$ra

$ra
Example of function call

A:
...  
jal B
...

B:
...

B2:
...  
jal C

C:
...  
jr $ra

B:
...

PC

$sp

saved $v*, $a*, $t*, $ra

B args

saved $s*
Example of function call

A: ...
 jal B
 ...

B: ...
 jal C

B2: ...
 jr $ra

C: ...
 jr $ra

PC

saved $v*, $a*, $t*, $ra

B args

saved $s*

C args

saved $v*, $a*, $t*, $ra

$sp
Example of function call

A: ...
   jal B
   ...

B: ...
   jal C
   B2: ...
   jr $ra

C: ...
   jr $ra
Example of function call

```
<table>
<thead>
<tr>
<th>saved</th>
<th>$v^<em>,$a^</em>, $t^*,$ra</th>
</tr>
</thead>
<tbody>
<tr>
<td>B args</td>
<td></td>
</tr>
<tr>
<td>saved</td>
<td>$s^*$</td>
</tr>
<tr>
<td>saved</td>
<td>$v^<em>,$a^</em>, $t^*,$ra</td>
</tr>
<tr>
<td>C args</td>
<td></td>
</tr>
<tr>
<td>saved</td>
<td>$s^*$</td>
</tr>
</tbody>
</table>

A: ...
    jal B
    ...

B: ...
    jal C

B2: ...
    jr $ra

C: ...
    jr $ra
```
Example of function call

A: ...
jal B
...

B: ...
jal C
B2: ...
jr $ra

C: ...
jr $ra
Example of function call

$sp$

<table>
<thead>
<tr>
<th>saved</th>
<th>$v^<em>, a^</em>, t^*, ra$</th>
</tr>
</thead>
<tbody>
<tr>
<td>B args</td>
<td></td>
</tr>
<tr>
<td>saved</td>
<td>$s^*$</td>
</tr>
<tr>
<td>saved</td>
<td>$v^<em>, a^</em>, t^*, ra$</td>
</tr>
<tr>
<td>C args</td>
<td></td>
</tr>
<tr>
<td>saved</td>
<td>$s^*$</td>
</tr>
</tbody>
</table>

A:
... 
jal B 
...
jr $ra$

PC

B:
... 
jal C

B2:
... 
jr $ra$

C:
... 
jr $ra$
Exceptions (software interrupts)

- MIPS terminology:
  - **Exception** = unexpected change in control flow caused *internally* (e.g., arithmetic overflow, undefined instruction) or a desired change of control yielded to OS (e.g., system call)
  - **Interrupt** = unexpected change in control flow caused *externally* (e.g., request from an I/O device)

- MIPS exceptions and interrupts are handled by a peripheral device “coprocessor 0” (cp0)
Coprocessor 0

- MIPS exceptions and interrupts are handled by “coprocessor 0” (cp0)
- cp0 has several 32-bit registers
  - **EPC** (cp0 register $14) – exception program counter
  - **Cause register** (cp0 register $13) – contains bits identifying exception cause
  - **Status register** (cp0 register $12) – is used to configure interrupts
- cp0 registers cannot be directly accessed by MIPS instructions
- Special instructions transfer information from/to cp0 like load/store:
  - **mfc0** $c0rt, $rd  
    *Move from coprocessor 0*
    Semantics: rd ← c0rt
  - **mtc0** $c0rd, $rt  
    *Move to coprocessor 0*
    Semantics: $c0rd ← rt
MIPS exception handling

- **Cause register** (alias $cr):
- **Exception code** (bits 0-6): describe the cause of the last exception
  
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>INT</td>
</tr>
<tr>
<td>6</td>
<td>IBUS</td>
</tr>
<tr>
<td>8</td>
<td>SYSCALL</td>
</tr>
<tr>
<td>12</td>
<td>OVF</td>
</tr>
</tbody>
</table>
- **Pending interrupts** (bits 8-15): when an interrupt at a given SW/HW level is raised, the corresponding bit is asserted
- **Branch delay** (bit 31): set to one if last exception occurred in an instruction scheduled in a delay slot of a branch
MIPS exception handling

- **Status register** (alias $sr): 

- **Interrupt enable** (bit 0): interrupts are not handled and EPC is not updated when 0

- **Interrupt mask** (bits 8-15): defining *masks* for the eight interrupt levels

- If an interrupt/exception occurs when its mask is set to 0, it is **ignored** though pending interrupt bit is raised
Exception vs function call

- When a **function** is called via `jal`:
  - Caller saves state prior to issuing `jal` so callee can safely modify registers
  - Control is transferred by setting PC to the address provided by the instruction
  - Return address to next instruction (PC+4) is saved to `$ra`
  - Function returns via `jr $ra`
Exception vs function call

- **Exception/interrupt** have no explicit call:
  - Control is transferred to a **fixed** location (usually 0x80000080) where the exception handler **must** reside
  - EPC stores the return address (PC+4)
  - Status register IE bit is masked to disable interrupts
  - If control is returned to the program, special instruction `rfe` (return from exception) restores the status register reenabling interrupts
- Exception is an unexpected event and exception handler shall take care of saving and restoring previous state
- Exception handlers may use registers $26-$27 (alias $k0-$k1) that are not used by user programs
Exception vs function call

- Typical exception return code

```
    mfc0 $k0, $epc
    rfe
    jr $k0  # issued in jr delay slot
```