Program Counter and GPRs (especially $sp, $ra, and $fp)

REVIEW OF RELEVANT CONCEPTS
Review of Program Counter

• PC is used to fetch an instruction
  – PC contains the address of the next instruction
  – The value in the PC is placed on the address bus and the memory is told to read
  – The PC is incremented, and the process is repeated for the next instruction

![Diagram of processor and memory](image)
### GPR's Used for Subroutine Support

<table>
<thead>
<tr>
<th>Assembler Name</th>
<th>Reg. Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>$0</td>
<td>Constant 0 value</td>
</tr>
<tr>
<td>$at</td>
<td>$1</td>
<td>Assembler temporary</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>$2-$3</td>
<td>Procedure return values or expression evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>$4-$7</td>
<td>Arguments/parameters</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>$8-$15</td>
<td>Temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>$16-$23</td>
<td>Saved Temporaries</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>$24-$25</td>
<td>Temporaries</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>$26-$27</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>$gp</td>
<td>$28</td>
<td>Global Pointer (Global and static variables/data)</td>
</tr>
<tr>
<td>$sp</td>
<td>$29</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>$30</td>
<td>Frame Pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>$31</td>
<td>Return address for current procedure</td>
</tr>
</tbody>
</table>
Subroutines (Functions)

- Subroutines are portions of code that we can call from anywhere in our code, execute that subroutine, and then **return to where we left off**

```c
void main() {
    ...
    x = 8;
    res = avg(x, 4);
    ...
}

int avg(int a, int b) {
    return (a+b)/2;
}
```

We call the subroutine to calculate the average and return to where we called it

A subroutine to calculate the average of 2 numbers
Subroutines

- Subroutines are similar to branches where we jump to a new location in the code

```c
void main() {
    ...
    x = 8;
    res = avg(x, 4);
    ...
}
int avg(int a, int b){
    return (a+b)/2;
}
```

Call “avg” sub-routine will require us to branch to that code
Normal Branches vs. Subroutines

• Difference between normal branches and subroutines branches is that with subroutines we have to return to where we left off

• We need to leave a link to the return location before we jump to the subroutine...once in the function it's too late

C code:

```c
void main() {
    ...
    x = 8;
    res = avg(x, 4);
    ...
}

int avg(int a, int b){
    return (a+b)/2;
}
```

1. Call “avg” sub-routine to calculate the average
2. After subroutine completes, return to the statement in the main code where we left off
Implementing Subroutines

• To implement subroutines in assembly we need to be able to:
  – Branch to the subroutine code
  – Know where to return to when we finish the subroutine

C code:

```
...  
res = avg(x, 4);  
...  

int avg(int a, int b) { ... }
```

Assembly:

```
.text
  ...
jal AVG
  ...

AVG:
  ...
jr $ra
```
Jumping to a Subroutine

• JAL instruction (Jump And Link)
  – Format: \texttt{jal Address/Label}
  – Similar to jump where we load an address into the PC [e.g. \texttt{PC = addr}]
    • Same limitations (26-bit address) as jump instruction
    • \texttt{Addr} is usually specified by a label

• JALR instruction (Jump And Link Register)
  – Format: \texttt{jalr $rs}
  – Jumps to address specified by $rs (so we can jump a full 32-bits)

• In addition to jumping, JAL/JALR stores the PC into R[31] ($ra = return address) to be used as a link to return to after the subroutine completes
Jumping to a Subroutine

• Use the JAL instruction to jump execution to the subroutine and leave a link to the following instruction.

Assembly:

```
0x400000  jal AVG
0x400004  add
...
AVG: = 0x400810
0x400810  add
...
jr $ra
```

JAL will cause the program to jump to the label AVG and store the return address in $ra/$31.
Returning from a Subroutine

- Use a JR with the $ra register to return to the instruction after the JAL that called this subroutine

PC before exec. of jr:

0040 08ec

$ra before exec. of jr:

0040 0004

PC after exec. of jr:

0040 0004

Go back to where we left off using the return address stored by JAL

jal will cause the program to jump to the label AVG and store the return address in $ra/$31.

```
0x400000 jal AVG
0x400004 add ...
AVG: = 0x400810
add ...
0x4008ec jr $ra
```
Return Addresses

- No single return address for a subroutine since AVG may be called many times from many places in the code
- JAL always stores the address of the instruction after it (i.e. PC of ‘jal’ + 4)

Assembly:

```
0x400000    jal AVG
0x400004    add...
0x400024    jal AVG
0x400028    sub...
0x400810    ...
```

0x400004 is the return address for this JAL
0x400028 is the return address for this JAL
Return Addresses

- A further complication is nested subroutines (a subroutine calling another subroutine)
- Example: Main routine calls SUB1 which calls SUB2
- Must store both return addresses but only one $ra register

Assembly:

```
...  
jal SUB1  
0x40001A ...
SUB1  
jal SUB2  
0x400208 jr $ra  
SUB2  
...  
jr $ra  
```
Dealing with Return Addresses

- Multiple return addresses can be spilled to memory
  - “Always” have enough memory
- Note: Return addresses will be accessed in reverse order as they are stored
  - 0x400208 is the second RA to be stored but should be the first one used to return
  - A stack is appropriate!

Assembly:

```
...  
jal SUB1
0x40001A ...

SUB1  jal SUB2
0x400208 jr $ra

SUB2  ...
       jr $ra
...  
```
Stacks

• Stack is a data structure where data is accessed in reverse order as it is stored (a.k.a. LIFO = Last-in First-out)
• Use a stack to store the return addresses and other data
• System stack defined as growing towards smaller addresses
  – MARS starts stack at 0x7fffeffc
  – Normal MIPS starts stack at 0x80000000
• Top of stack is accessed and maintained using $sp=R[29]$ (stack pointer)
  – $sp$ points at top occupied location of the stack
Stacks

- 2 Operations on stack
  - Push: Put new data on top of stack
    - Decrement $sp$
    - Write value to where $sp$ points
  - Pop: Retrieves and “removes” data from top of stack
    - Read value from where $sp$ points
    - Increment $sp$ to effectively “delete” top value
Push Operation

- Recall we assume $sp$ points at top **occupied** location
- Push: Put new data on top of stack
  - Decrement SP
    - addi $sp,$sp,-4
    - Always decrement by 4 since addresses are always stored as words (32-bits)
  - Write return address ($ra$) to where SP points
    - sw $ra, 0($sp)

Push return address (e.g. 0x00400208)

- $sp = \text{00400208}$
- Decrement SP by 4 (since pushing a word), then write value to where $sp$ is now pointing
Pop Operation

- Pop: Retrieves and "removes" data from top of stack
  - Read value from where SP points
    - lw  $ra, 0($sp)
  - Increment SP to effectively "deletes" top value
    - addi  $sp,$sp,4
    - Always increment by 4 when popping addresses

**Warning:** Because the stack grows towards lower addresses, when you push something on the stack you subtract 4 from the SP and when you pop, you add 4 to the SP.
Subroutines and the Stack

- When writing native assembly, programmer must add code to manage return addresses and the stack
- At the beginning of a routine (PREAMBLE)
  - Push $ra (produced by 'jal') onto the stack
    \[
    \text{addi } \$sp,\$sp,-4 \\
    \text{sw } \$ra,0(\$sp)
    \]
- Execute subroutine which can now freely call other routines
- At the end of a routine (POSTAMBLE)
  - Pop/restore $ra from the stack
    \[
    \text{lw } \$ra,0(\$sp) \\
    \text{addi } \$sp,\$sp,4 \\
    \text{jr } \$ra
    \]
Subroutines and the Stack

... 
jal SUB1

0x40001A ... 

SUB1
0 addi $sp,$sp,-4
sw $ra,0($sp)
jal SUB2
0x400208 lw $ra,0($sp)
addi $sp,$sp,4
jr $ra

SUB2
addi $sp,$sp,-4
sw $ra,0($sp)
... 
lw $ra,0($sp)
addi $sp,$sp,4
jr $ra

0 $sp = 7fffeffc 0000 0000 7fffeffc
$ra = 0040001a 0000 0000 7fffeff8

1 $sp = 7fffeff8 0000 0000 7fffeffc
$ra = 0040001a 0000 0000 7fffeff4

2 $sp = 7fffeff4 0000 0000 7fffeffc
$ra = 00400208 0040 0208 7fffeff4

3 $sp = 7fffeffc 0000 0000 7fffeffc
$ra = 0040001a 0040 0208 7fffeff4
Optimizations for Subroutines

• Definition:
  – Leaf procedure: A procedure that does not call another procedure

• Optimization
  – A leaf procedure need not save $ra onto the stack since it will not call another routine (and thus not overwrite $ra)
Leaf Subroutine

...  
jal SUB1  
0x40001A ...

SUB1

0 addi $sp,$sp,-4  
sw $ra,0($sp)  
jal SUB2  
0x400208 lw $ra,0($sp)  
addi $sp,$sp,4  
3 jr $ra

// Leaf Procedure

SUB2

2 jr $ra

0 $sp = 7ffeffe  
1 $ra = 0040001a  
2 $ra = 00400208  
3 $ra = 0040001a
Using the stack for passing arguments, saving registers, & local variables

STACK FRAMES
Arguments and Return Values

- Most subroutine calls pass arguments/parameters to the routine and the routine produces return values.
- To implement this, there must be locations agreed upon by caller and callee for where this information will be found.
- MIPS convention is to use certain registers for this task:
  - $a0 - $a3 ($4 – $7) used to pass up to 4 arguments
  - $v0, $v1 ($2, $3) used to return up to a 64-bit value

```c
void main() {
    int arg1, arg2;
    ans = avg(arg1, arg2);
}

int avg(int a, int b) {
    int temp=1;  // local var’s
    return a+b >> temp;
}
```
Arguments and Return Values

- Up to 4 arguments can be passed in $a0-$a3
  - If more arguments, use the stack
- Return value (usually HLL’s) limit you to one return value in $v0
  - For a 64-bit return value, use $v1 as well

```
MAIN:  li   $a0, 5
   li   $a1, 9
   jal  AVG
   sw   $v0, ($s0)
   ...  
   lw   $a0, 0($s0)
   li   $a1, 0($s1)
   jal  AVG
   sw   $v0, ($s0)
   ...

AVG:   li   $t0, 1
   add  $v0, $a0, $a1
   srav $v0, $v0, $t0
   jr    $ra
```
Assembly & HLL’s

• When coding in assembly, a programmer can optimize usage of registers and store only what is needed to memory/stack
  – Can pass additional arguments in registers (beyond $a0-$a3)
  – Can allocate variables to registers (not use memory)

• When coding in an high level language & using a compiler, certain conventions are followed that may lead to heavier usage of the stack
  – We have to be careful not to **overwrite** registers that have useful data
Compiler Handling of Subroutines

• High level languages (HLL) use the stack:
  – for storage of local variables declared in the subroutine
  – to save register values including the return address
  – to pass additional arguments to a subroutine

• Compilers usually put data on the stack in a certain order, which we call a stack frame
Stack Frames

- **Frame = Def:** All data on stack belonging to a subroutine/function
  - Space for local variables (those declared in a function)
  - Space for saved registers ($ra and others)
  - Space for arguments (in addition to $a0-$a3)

```c
void main() {
    int ans, x, y;
    ...
    ans = avg(x, y);
}
int avg(int a, int b) {
    int temp=1; // local var’s
    ...
}
```

Stack Frame Organization

- **Main Routine’s Stack Frame**
  - Local Vars. (ans, x, y)
  - Saved Regs.
  - Subroutine’s arguments (copy of x,y)

- **AVG's Stack Frame**
  - Local Vars. (temp)
  - Saved Regs. ($ra,$fp, others)
  - Sub-subr’s arguments

Stack Growth
Accessing Values on the Stack

- Stack pointer ($sp$) is usually used to access only the top value on the stack.
- To access arguments and local variables, we need to access values buried in the stack:
  - We can simply use an offset from $sp$ [ 8($sp$) ]
- Unfortunately other push operations by the function may change the $sp$ requiring different displacements at different times for the same variable:
  - For now this is fine, but a compilers class would teach you alternate solutions.

```
lw $t0, 16($sp)  # access var. temp
addi $sp, $sp, -4  # $sp changes
sw $t0, 20($sp)  # access temp with  # diff. offset
```

To access parameters we could try to use some displacement [i.e. d($sp$) ] but if $sp$ changes, must use new offset value.
Local Variables

- A function's **local variables** are allocated on the stack

```c
void main() {
    // Allocate 3 integers
    int ans, arg1=5, arg2=7;
    ans = avg(arg1, arg2);
} // vars. deallocated here
```

**C Code**

**Equivalent Assembly**

```
MAIN:     addi $sp, $sp, -4
sw $ra, 0($sp) # save $ra
    # Now allocate 3 integers
addi $sp, $sp, -12
li $t0, 5
sw $t0, 4($sp)
li $t0, 7
sw $t0, 8($sp)
    ...
jal AVG     # call function
sw $v0, 0($sp) #store ans.
    ...
    # deallocate local vars
addi $sp,$sp, 12
lw $ra, 0($sp)
addi $sp,$sp, 4
jr $ra
```
Local Variables

- Locally declared arrays are also allocated on the stack
- **Be careful**: variables and arrays often must start on well-defined address boundaries

```c
void main() {
    char mystr[14] = "hello...";
    double z;
}
```

**C Code**

**Equivalent Assembly**

```
MAIN:  addi $sp, $sp, -4
       sw $ra, 0($sp)  # save $ra
       # Now allocate array
       addi $sp, $sp, -16  # not -14
       # May pad to get to 8-byte
       # boundary..
       # now alloc. z
       addi $sp, $sp, -8

       # deallocate local vars
       addi $sp, $sp, 24
       lw $ra, 0($sp)
       addi $sp, $sp, 4
       jr $ra
```
Saved Registers Motivation

• Assume the following C code
• Now assume each function was written by a different programmer on their own (w/o talking to each other)
• What could go wrong?

```c
int x = 5, nums[10];
int main()
{ caller(x, nums);
  return 0;
}
int caller(int z, int* dat)
{ int a = dat[0] + 9;
  return a + dat[3];
}
```

Caller wants to use $s0$ but what if main has a value in $s0$ that will be needed later
Solution

• If you're not sure whether some other subroutine is using a register (and needs it later)...
  
  • **Push** it to the stack before you overwrite it
    – Recall a push:
      
      \[
      \text{addi } \$sp, \$sp, -4 \\
      \text{sw } \text{reg\_to\_save}, 0(\$sp)
      \]

  • **Pop** it from the stack before you return
    – Recall a pop:
      
      \[
      \text{lw } \text{reg\_to\_restore}, 0(\$sp) \\
      \text{addi } \$sp, \$sp, 4
      \]
Solution

- If you're not sure whether some other subroutine is using a register (and needs it later)...

### Stack Diagrams

**At start of caller()**

- $sp: \text{7fffeffc}$
- $ra: \text{0040208c}$
- $s0: \text{10001230}$

### After li $s0, 9$

- $sp: \text{7fffeff4}$
- $ra: \text{0040208c}$
- $s0: \text{00000009}$

### Before 'jr $ra$'

- $sp: \text{7fffeffc}$
- $ra: \text{0040208c}$
- $s0: \text{10001230}$

### Code Snippet

```
.MAIN:
    la $s0, x
    lw $a0, 0($s0)
    la $a1, NUMS
    jal CALLER
    sw $v0, 0($s0)
    ...

.CALLER:
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    addi $sp, $sp, -4
    sw $s0, 0($sp)
    li $s0, 9
    ...
    add $v0, $v0, $a0
    lw $s0, 0($sp)
    addi $sp, $sp, 4
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra
```
Summary

• To support subroutines 'jal' saves return address in $ra
• To support nested subroutines we need to save $ra values onto the stack
• The stack is a common memory location to allocate space for saved values and local variables