CSSE 232
Computer Architecture I

Procedures I

Class Status

Reading for today

- 2.8
- B.6

Outline

- Big immediates and `$at`
- Procedure steps
- Instructions
- Register review
- Spilling registers
- Stack and frames
- Examples

Big Immediates and `$at`

`lw $t1, A($t1)`

- Read the value from memory at address (`A` + `$t1` contents) and store result in register `$t1`.
- `lw` is an I-type instruction. I-types support 16 bit immediate values.
- How is `lw` handled if `A` is a 16 bit address?

Big Immediates and `$at`

`lw $t1, A($t1)`

- Read the value from memory at address (`A` + `$t1` contents) and store result in register `$t1`.
- `lw` is an I-type instruction. I-types support 16 bit immediate values.
- How is `lw` handled if `A` is a 16 bit address?
  - I-types support 16 bit immediate, so no problem
- What if `A` is a 32-bit address?

Big Immediates and $at

`lw $t1, A($t1)`

- Read the value from memory at address (`A` + `$t1` contents) and store result in register `$t1`.
- `lw` is an I-type instruction. I-types support 16 bit immediate values.
- How is `lw` handled if `A` is a 16 bit address?
  - I-types support 16 bit immediate, so no problem
- What if `A` is a 32-bit address?
  - I-types only support 16 bit immediates, so load in two steps
  - Load upper 16 bits with `lui`
  - Load lower 16 bits with `ori` or clever use of `lw`

Why do we need Procedures/Functions?

Why do we need Procedures/Functions?

- Breaks code into small sections
- Gives code defined boundaries
- More manageable
- Easier to modify
- Easier to maintain
- Reusable

Procedure calling

```
int main() {
  int a = 1; int b = 2;
  int c = add(a, b);
  return 2 * c;
}
```

```
int add(int x, int y) {
  return x + y;
}
```

- Steps required
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call

Procedure Call Instructions

- Procedure call: jump and link
  `jal ProcedureLabel`
  - Address of following instruction put in `$ra`
  - Jumps to target address

- Procedure return: jump register
  `jr $ra`
  - Sets the address in `$ra` as the next instruction

Procedure Call Instructions

- `jal ProcedureLabel`: jump and link
  - Wipes out `$ra`, puts a new value in (new return address)
  - Old return address is lost!

- What should we do?

Procedure Call Instructions

- `jal ProcedureLabel`: jump and link
  - Wipes out `$ra`, puts a new value in (new return address)
  - Old return address is lost!
- What should we do?
- Save return address somewhere...

Procedure Call Instructions

- `jal ProcedureLabel`: jump and link
  - Wipes out `$ra`, puts a new value in (new return address)
  - Old return address is lost!

- What should we do?

- Save return address somewhere...
- Stack would probably be good

Program Counter (PC)

Special register which holds the address of the **next** instruction

The `jal` instruction saves **PC + 4** in `$ra`

System and Call Registers

Register #	Register Name	Description
0	zero	Hardwired to zero
1	at	Reserved for assembler
2	v0	Return values from procedure calls
3	v1	
4	a0	Arguments passed to procedure calls
5	a1	
6	a2	
7	a3	

Temporary Registers

Register #	Register Name	Description
8	t0	Temporary values, caller saves
9	t1	
10	t2	
11	t3	
12	t4	
13	t5	
14	t6	
15	t7	

Save Registers

Register #	Register Name	Description
16	s0	Saved values, callee saves
17	s1	
18	s2	
19	s3	
20	s4	
21	s5	
22	s6	
23	s7	

Temporary and System Registers

Register #	Register Name	Description
24	t8	Temporary values caller saves
25	t9	
26	k0	Reserved for OS kernel
27	k1	
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	Frame pointer
31	ra	Return address

Register Use

- MIPS has 10 `$t` registers, 8 `$s` registers
- What if a program needs more than 18 registers?

Register Use

- MIPS has 10 `$t` registers, 8 `$s` registers
- What if a program needs more than 18 registers?
    - Store in memory when not in use (spilling registers)

Register Use

- MIPS has 10 `$t` registers, 8 `$s` registers
- What if a program needs more than 18 registers?
    - Store in memory when not in use (spilling registers)

- What if a program uses all 18 registers, then calls a procedure?
- Can that procedure only use the `$a`*n* and `$v`*n* registers?

Register Use

- MIPS has 10 `$t` registers, 8 `$s` registers
- What if a program needs more than 18 registers?
  - Store in memory when not in use (spilling registers)

- What if a program uses all 18 registers, then calls a procedure?
- Can that procedure only use the `$a`*n* and `$v`*n* registers?
  - Save caller's registers in memory?

Register Use

- MIPS has 10 `$t` registers, 8 `$s` registers
- What if a program needs more than 18 registers?
    - Store in memory when not in use (spilling registers)

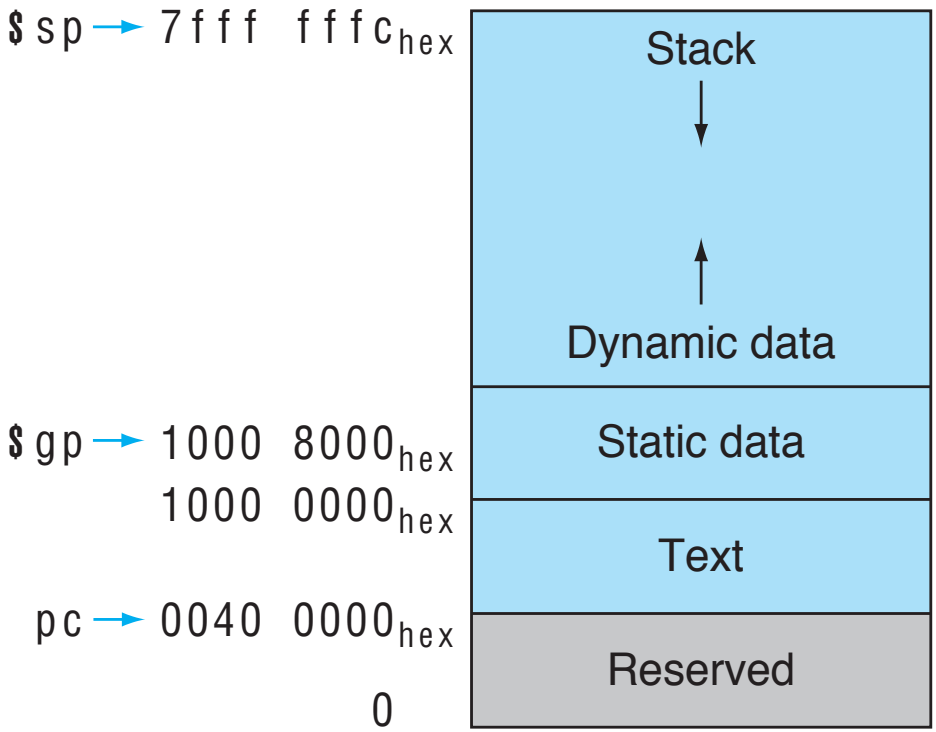
- What if a program uses all 18 registers, then calls a procedure?
- Can that procedure only use the `$a`*n* and `$v`*n* registers?
    - Save caller's registers in memory?

- We can define a 'stack' of memory to save registers

Spilling registers

- Stack
  - Push
  - Pop
- Stack pointer (register 29)
- Grow from higher addresses to lower addresses
  - Push values : subtract from stack pointer!!!
  - Pop values : add to stack pointer!!!

Stack Layout



Stack Frames

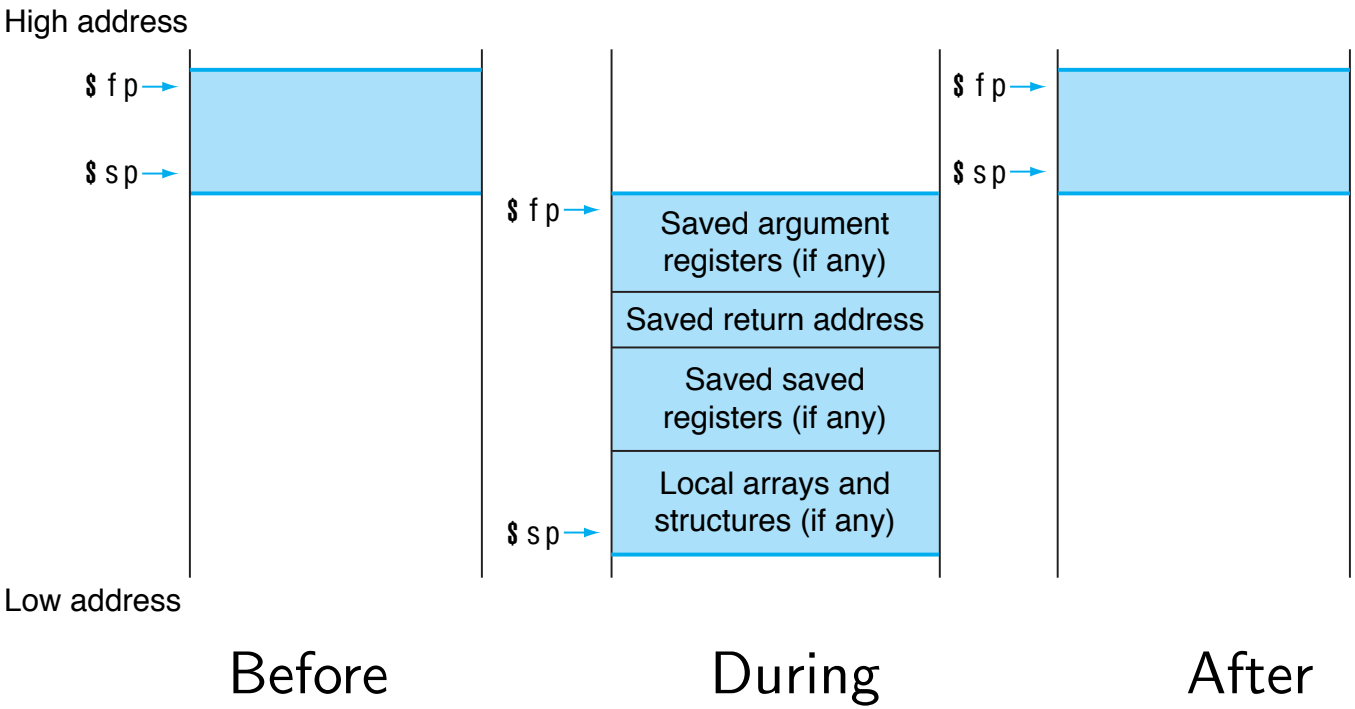
- Also called an activation record or procedure frame
- Segment of stack containing a procedure's saved registers and local variables
- Also used for extra arguments
- Frame pointer (`$fp`) points to the first word of the frame of a procedure
- Stack pointer (`$sp`) and frame pointer (`$fp`) define the bounds of the stack frame

Argument conventions

- If the procedure takes four or less arguments
  - Place arguments in `$a0`-`$a3`
- If the procedure takes more than four arguments
  - Place first four arguments in `$a0`-`$a3`
  - Place extra arguments on stack in order
  - Procedure uses `$sp` to locate extra arguments

This is a simplified convention, actual MIPS programs use a more complex convention.

Stack Allocation During Call



Register Use

- Caller function uses `$t`*n* and `$s`*n* registers
- Callee function also uses `$t`*n* and `$s`*n* registers
- Must avoid overwriting other procedure's register data
  - Can save register values on stack
  - Use register for whatever is needed
  - Restore value when done using

Register Use

- Backup `$s` registers before using
- Restore `$s` registers before returning to caller
  - Caller should never notice any changes!
- Never assume `$t` registers are valid across calls
- Backup if needed (on stack)

Call conventions

- When procedure begins:
  - Save `$s`*n* before using
- Before making a call (i.e. before using `jal`):
  - Save `$ra`
  - Save `$t`*n*, `$a`*n*, and `$v`*n* if needed
- After making a call:
  - Restore `$ra`
  - Restore `$t`*n*, `$a`*n*, and `$v`*n* if needed
- Before returning to caller:
  - Restore `$s`*n* if used
  - Restore `$sp` before returning (i.e. before using `jr $ra`)

Procedure Call

```
int main() {
  int w, x, y;
  ...  //put values in w and x
  y = leaf_example(w, x);
  y = w + y;
  ...
}
```

Assume w, x are stored in `$t0`,`$t1` and y is stored in `$s0`.

Procedure Call - just the call

```
int main() {
  int w, x, y;
  ...  //put values in w and x
  y = leaf_example(w, x);
  y = w + y;
  ...
}
```

```
  main:
  ... #put a value in x
  addi $a0, $t0, 0   #put w in arg reg
  addi $a1, $t1, 0   #put x in arg reg
  jal  leaf_example  #make procedure call
  addi $s0, $v0, 0   #put return value in y
  add  $s0, $t1, $s0 #compute new y
  ...
  jr $ra
```

Assume w, x are stored in $t0,$t1 and y is stored in $s0.

Procedure Call - just the call

```
int main() {
  int w, x, y;
  ...  //put values in w and x
  y = leaf_example(w, x);
  y = w + y;
  ...
}
```

```
  main:
  ... #put a value in x
  addi $a0, $t0, 0   #put w in arg reg
  addi $a1, $t1, 0   #put x in arg reg
  jal  leaf_example  #make procedure call
  addi $s0, $v0, 0   #put return value in y
  add  $s0, $t1, $s0 #compute new y
  ...
  jr $ra
```

First try... still need to save $t1...

Assume w, x are stored in $t0,$t1 and y is stored in $s0.

Procedure Call - save $t0

```
int main() {
  int w, x, y;
  ...  //put values in w and x
  y = leaf_example(w, x);
  y = w + y;
  ...
}
```

Assume w, x are stored in $t0,$t1 and y is stored in $s0.

```
main:
... #put a value in x
addi $sp, $sp, -4 #adjust stack to save a value
sw   $t0, 4($sp)  #save t0 for later
addi $a0, $t0, 0  #put w in arg reg
addi $a1, $t1, 0  #put x in arg reg
jal  leaf_example  #make procedure call
lw   $t0, 4($sp)  #restore t0
addi $s0, $v0, 0  #put return value in y
add  $s0, $t1, $s0 #compute new y
...
addi $sp, $sp, 4  #restore stack pointer
jr $ra
```

Procedure Call - save $t0

```
int main() {
  int w, x, y;
  ...  //put values in w and x
  y = leaf_example(w, x);
  y = w + y;
  ...
}
```

Assume w, x are stored in $t0,$t1 and y is stored in $s0.

```
main:
... #put a value in x
addi $sp, $sp, -4 #adjust stack to save a value
sw   $t0, 4($sp)  #save t0 for later
addi $a0, $t0, 0  #put w in arg reg
addi $a1, $t1, 0  #put x in arg reg
jal  leaf_example #make procedure call
lw   $t0, 4($sp)  #restore t0
addi $s0, $v0, 0  #put return value in y
add  $s0, $t1, $s0 #compute new y
...
addi $sp, $sp, 4  #restore stack pointer
jr $ra
```

Now, add the $s0 save/restore...

Procedure Call - save $s0

```
int main() {
  int w, x, y;
  ...  //put values in w and x
  y = leaf_example(w, x);
  y = w + y;
  ...
}
```

Assume w, x are stored in $t0,$t1 and y is stored in $s0.

```
  main:
  ... #put a value in x
  addi $sp, $sp, -8 #adjust stack to save 2 values
  sw   $s0, 0($sp)   #save s0 before using
  sw   $t0, 4($sp)   #save t0 for later
  addi $a0, $t0, 0   #put w in arg reg
  addi $a1, $t1, 0   #put x in arg reg
  jal  leaf_example  #make procedure call
  lw   $t0, 4($sp)   #restore t0
  addi $s0, $v0, 0   #put return value in y
  add  $s0, $t1, $s0 #compute new y
  ...
  lw   $s0, 0($sp)   #restore s0
  addi $sp, $sp, 8  #restore stack pointer
  jr $ra
```

Procedure Call - save $s0

```
int main() {
  int w, x, y;
  ...  //put values in w and x
  y = leaf_example(w, x);
  y = w + y;
  ...
}
```

Assume w, x are stored in $t0,$t1 and y is stored in $s0.

```
  main:
  ... #put a value in x
  addi $sp, $sp, -8 #adjust stack to save 2 values
  sw   $s0, 0($sp)   #save s0 before using
  sw   $t0, 4($sp)   #save t0 for later
  addi $a0, $t0, 0   #put w in arg reg
  addi $a1, $t1, 0   #put x in arg reg
  jal  leaf_example  #make procedure call
  lw   $t0, 4($sp)   #restore t0
  addi $s0, $v0, 0   #put return value in y
  add  $s0, $t1, $s0 #compute new y
  ...
  lw   $s0, 0($sp)   #restore s0
  addi $sp, $sp, 8  #restore stack pointer
  jr $ra
```

Finally, let's add in the $ra save...

Procedure Call - save $ra

```
int main() {
  int w, x, y;
  ...  //put values in w and x
  y = leaf_example(w, x);
  y = w + y;
  ...
}
```

Assume w, x are stored in $t0,$t1 and y is stored in $s0.

```
  main:
  ... #put a value in x
  addi $sp, $sp, -12 #adjust stack to save 3 values
  sw   $s0, 0($sp)   #save s0 before using
  sw   $t0, 4($sp)   #save t0 for later
  sw   $ra, 8($sp)   #save ra before jump
  addi $a0, $t0, 0   #put w in arg reg
  addi $a1, $t1, 0   #put x in arg reg
  jal  leaf_example  #make procedure call
  lw   $t0, 4($sp)   #restore t0
  addi $s0, $v0, 0   #put return value in y
  add  $s0, $t1, $s0 #compute new y
  ...
  lw   $s0, 0($sp)   #restore s0
  lw   $ra, 8($sp)   #restore ra
  addi $sp, $sp, 12  #restore stack pointer
  jr $ra
```

Procedure Body

```
int leaf_example(int a, int b)
{
  int c, d;
  c = 5;
  d = a + b +c;
  ...
  return d;
}
```

Where are a and b stored?
Assume c must be stored in $s0 and d is stored in $t0.

Procedure Body

```
int leaf_example(int a, int b)
{
  int c, d;
  c = 5;
  d = a + b +c;
  ...
  return d;
}
```

Where are a and b stored?
Assume c must be stored in $s0 and d is stored in $t0.

```
  Leaf_example:
  addi $sp, $sp, -4  #adjust stack for 1 value
  sw   $s0, 0($sp)   #place s0 contents on stack
  addi $s0, $zero, 5 #s0 gets 5
  add  $t0, $a0, $a1 #add arguments a + b, store in temp
  add  $t0, $t0, $s0 #add temp + c, store in s0
  ...
  addi $v0, $t0, 0   #move return to t0
  lw   $s0, 0($sp)   #restore s0
  addi $sp, $sp, 4   #restore stack pionter
  jr   $ra           #return to line after call
```

Questions?

- Big immediates and `$at`
- Procedure steps
- Instructions
- Register review
- Spilling registers
- Stack and frames
- Examples