ECE 550D
Fundamentals of Computer Systems and Engineering

Fall 2023

Instruction Set Architectures (ISAs) and MIPS

Xin Li & Dawei Liu
Duke Kunshan University

Slides are derived from work by
Andrew Hilton, Tyler Bletsch and Rabih Younes (Duke)

Last time…

- Who can remind us what we did last time?
  - MIPS ISA and Assembly Programming
  - More on Assembly Programming today

J-Type <op> immediate

- 16-bit imm limits to +/- 32K insns
- Usually fine, but sometimes need more…
- J-type insns provide long range, unconditional jump:

J-type: Jump / Call

- Specifies lowest 28 bits of PC
  - Upper 4 bits unchanged
  - Range: 64 Million instruction (256 MB)

- Can jump anywhere with `jr $reg` (jump register)

Remember our F2C program fragment?

- Consider the following C fragment:

```
int tempF = 87;              li   $3, 87
int a = tempF – 32;          addi $4, $3, -32
a = a * 5;                   li   $6, 5
                             mul  $4, $4, $6
int tempC = a / 9;           li   $6, 9
                             div  $5, $4, $6
```

- If we were really doing this…

  We would write a **function** to convert f2c and call it

More likely: a function

- Like this:

```
int f2c (int tempF) {
   int a = tempF - 32;
   a = a * 5;
   int tempC = a / 9;
   return tempC;
}
…
…
int tempC = f2c(87);
```


Need a way to call f2c and return

- Call: Jump… but also remember where to go back
  - May be many calls to f2c() in the program
  - Need some way to know where we were
  - Instruction for this **jal**
  - `jal label`
    - Store PC +4 into register $31
    - Jump to label
- Return: Jump… back to wherever we were
  - Instruction for this **jr**
  - `jr $31`
  - Jump back to address stored by jal in $31

More likely: a function to convert

- Like this:

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}                              //jr $31
…
…
int tempC = f2c(87);           //jal f2c
```

- But that's not all…

More likely: a function to convert

- Like this:

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}                              //jr $31
…
…
int tempC = f2c(87);           //jal f2c
```

- Need to pass 87 as argument to f2c

More likely: a function to convert

- Like this:

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}                              //jr $31
…
…
int tempC = f2c(87);           //jal f2c
```

- Need to return `tempC` to caller

More likely: a function to convert

- Like this:

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}                              //jr $31
…
…
int tempC = f2c(87);           //jal f2c
```

- Also, may want to use same registers in multiple functions
  - What if f2c called something? Would re-use $31

Calling Convention

- All of these are reasons for a calling convention
  - Agreement of how registers are used
  - Where arguments are passed, results returned
  - Who must save what if they want to use it
  - Etc..

MIPS Register Usage/Naming Conventions

0	zero constant	
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		
15	t7	

16	s0	temporary: callee saves
. . .		
23	s7	
24	t8	temporary: 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

Also 32 floating-point registers: \$f0 .. \$f31

Important: The only general purpose registers are the \$s and \$t registers.

Everything else has a specific usage:
\$a = arguments, \$v = return values, \$ra = return address, etc.

Caller Saves

- Caller saves registers
  - If some code is about to call another function...
  - And it needs the value in a caller saves register ($t0,$t1...)
  - Then it has to save it on the **stack** before the call
  - And restore it after the call

Callee Saves

- Callee saves registers
  - If some code wants to use a callee saves register (at all)
  - It has to save it to the stack before it uses it
  - And restore it before it returns to its caller
  - But, it can assume any function it calls will not change the register
    - Either won't use it, or will save/restore it

More likely: a function to convert

```
int f2c (int tempF) {
   int a = tempF - 32;
   a = a * 5;
   int tempC = a / 9;
   return tempC;
}
```

```
f2c:
   addi $t0, $a0, -32
```

tempF is in $a0 by calling convention

More likely: a function to convert

```
int f2c (int tempF) {
   int a = tempF - 32;
   a = a * 5;
   int tempC = a / 9;
   return tempC;
}
```

```
f2c:
   addi $t0, $a0, -32
```

We can use $t0 for a temp (like a) without saving it

More likely: a function to convert

```
int f2c (int tempF) {
   int a = tempF - 32;
   a = a * 5;
   int tempC = a / 9;
   return tempC;
}
```

```
f2c:
   addi $t0, $a0, -32
   li $t1, 5
   mul $t0, $t0, $t1
   li $t1, 9
   div $t2, $t0, $t1
```


More likely: a function to convert


```
int f2c (int tempF) {
   int a = tempF - 32;
   a = a * 5;
   int tempC = a / 9;
   return tempC;
}
```


```
f2c:
   addi $t0, $a0, -32
   li $t1, 5
   mul $t0, $t0, $t1
   li $t1, 9
   div $t2, $t0, $t1
   addi $v0, $t2, 0
   jr $ra
```

More likely: a function to convert

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}
```

```
f2c:
   addi $t0, $a0, -32
   li $t1, 5
   mul $t0, $t0, $t1
   li $t1, 9
   div $t2, $t0, $t1
   addi $v0, $t2, 0
   jr $ra
```

A smart compiler would just do
div $v0, $t0, $t1

More likely: a function to convert

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}
...
...
int tempC = f2c(87)
```

```
f2c:
   addi $t0, $a0, -32
   li $t1, 5
   mul $t0, $t0, $t1
   li $t1, 9
   div $t2, $t0, $t1
   addi $v0, $t2, 0
   jr $ra
…
addi $a0, $0, 87
jal f2c
addi $t0, $v0, 0
```

What it would take to make SPIM happy

```
      .globl f2c    # f2c can be called from any file
      .text         # goes in "text" region
f2c:              #   (remember memory picture?)
addi $t0, $a0, -32
li $t1, 5
mul $t0, $t0, $t1
li $t1, 9
div $t2, $t0, $t1
addi $v0, $t2, 0
jr $ra

.end f2c        # end of this function
```

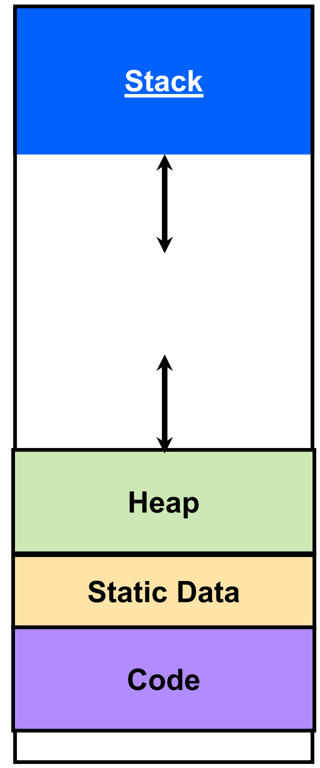

Assembly Language (cont.)

- Directives: tell the assembler what to do...
- Format "."<string> [arg1], [arg2] …

- Examples
  - .data — # start a data segment.
  - .text — # start a code segment.
  - .align n — # align segment on 2^n byte boundary.
  - .ascii <string> — # store a string in memory.
  - .asciiz <string> — # store a null terminated string in memory
  - .word w1, w2, . . . , wn — # store n words in memory.
  - .space n — # reserve n bytes of space

The Stack

- May need to use the stack for…
  - Local variables
  - Saving registers
    - Across calls
    - Spilling variables (not enough regs)
  - Passing more than 4 arguments

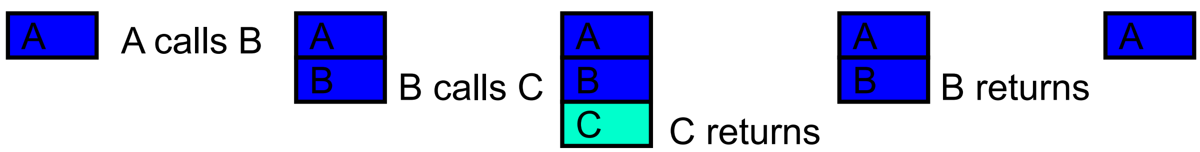


Stack Layout

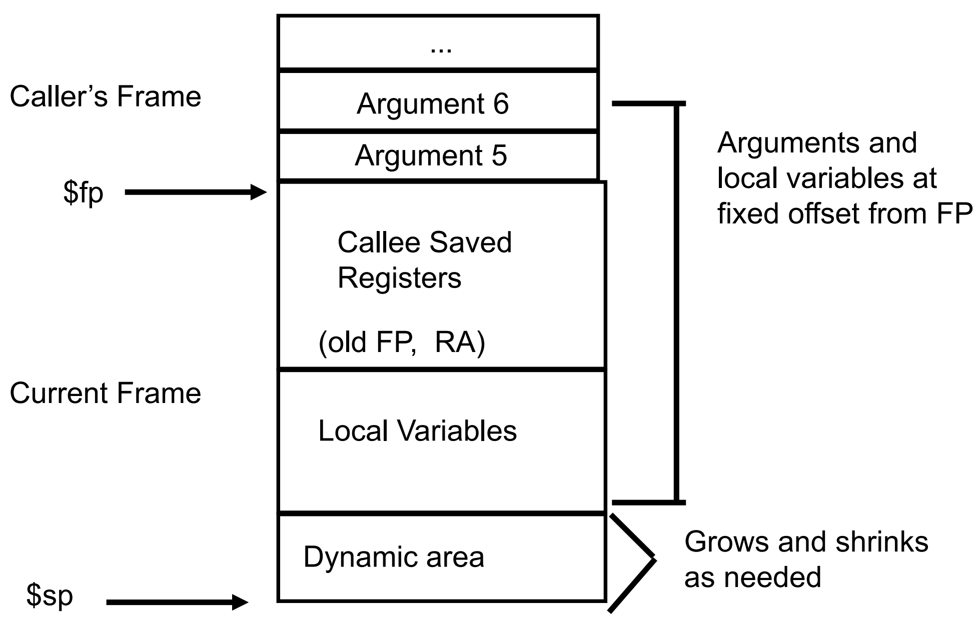
- Stack is in memory:
  - Use loads and stores to access
  - But what address to load/store?
- Two registers for stack:
  - Stack pointer ($sp): Points at end (bottom) of stack
  - Frame pointer ($fp): Points at top of current stack frame

Procedures Use the Stack

- In general, procedure calls obey **stack discipline**
  - Local procedure state contained in **stack frame**
  - Where we can save registers
  - When a procedure is called, a new frame opens
  - When a procedure returns, the frame collapses
- Procedure stack is **in memory**
  - Starts at "top" of memory and grows down



Call-Return Linkage: Stack Frames



MIPS/GCC Procedure Calling Conventions

Calling Procedure

- Step-1: Setup the arguments:
  - The first four arguments (arg0-arg3) are passed in registers \$a0-\$a3
  - Remaining arguments are pushed onto the stack
    (in reverse order, arg5 is at the bottom of the stack)
- Step-2: Save caller-saved registers
  - Save registers \$t0-\$t9 if they contain live values at the call site.
- Step-3: Execute a jal instruction.
- Step-4: Cleanup stack by updating \$sp (if more than 4 args)

MIPS/GCC Procedure Calling Conventions (cont.)

Called Routine (if any frame is needed)

- Step-1: Establish stack frame.
  - Subtract the frame size from the stack pointer.
    subiu \$sp, \$sp, <frame-size>
  - Typically, minimum frame size is 32 bytes (8 words).
- Step-2: Save callee saved registers in the frame.
  - Register \$fp is always saved.
  - Register \$ra is saved if routine makes a call.
  - Registers \$s0-\$s7 are saved if they are used.
- Step-3: Establish Frame pointer
  - Add the stack <frame size> - 4 to the address in \$sp
    addiu \$fp, \$sp, <frame-size> - 4

Frame pointer isn't strictly necessary, but helps with debugging.
We'll see examples with and without it.

MIPS/GCC Procedure Calling Conventions (cont.)

On return from a call

- Step-1: Put returned values in registers $v0, [$v1].
  (if values are returned)
- Step-2: Restore callee-saved registers.
  - Restore $fp and other saved registers. [$ra, $s0 - $s7]
- Step-3: Pop the stack
  - Add the frame size to $sp.
    addiu $sp, $sp, <frame-size>
- Step-4: Return
  - Jump to the address in $ra.
    jr $ra

Summary

- MIPS ISA and Assembly Programming
- Continue on next lecture…