Lecture 4: RISC-V Instruction Set, Part 3

CS10014 Computer Organization

Department of Computer Science
Tsung Tai Yeh
Thursday: 1:20 pm– 3:10 pm
Classroom: EC-022

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CS 252 at UC Berkeley
    - https://people.eecs.berkeley.edu/~culler/courses/cs252-s05/
  - CSCE 513 at University of South Carolina
    - https://passlab.github.io/CSCE513/

Outline

- Branch instructions
- Bitwise/Logical instructions
- Loops in assembly
- Inequality in RISC-V
- Functional calling

C Decisions: if Statements

- 2 kinds of if statements in C
  - if (condition) clause
  - if (condition) clause1 else clause2
- Rearrange 2nd if into following:

```
if   (condition) goto L1;
     clause2;
     goto L2;
L1:  clause1;

L2:
```

RISC-V Decision Instructions

- **Conditional branches**
- Decision instruction in RISC-V
  - beq register1, register2, L1
  - beq is "Branch if (registers are) equal"

    if (register1 == register2) goto L1
- Complementary RISC-V decision instruction
  - bne register1, register2, L1
  - bne is "Branch if (registers are) not equal"

    if (register != register2) goto L1

RISC-V Goto Instruction

- **Unconditional branches**

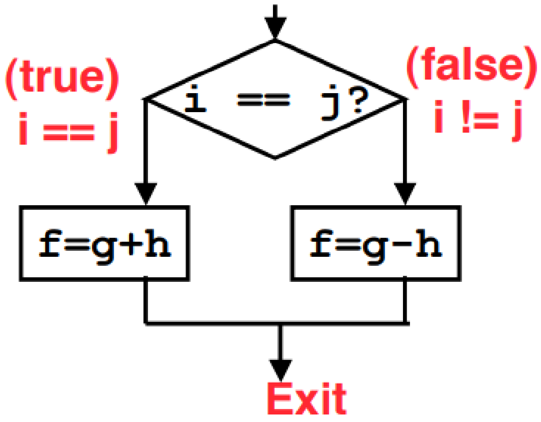
  j label
- Call a jump instruction
  - Jump (or branch) directly to the given label without needing to satisfy any condition
  - Same meaning as (using C)
    - goto label
  - Technically, it’s the same as
    - beq $x0, $x0, label

    Since it always satisfies the condition

Compiling C if into RISC-V (1/2)

- **Compile by hand**

```
if (i == j) f=g+h;
else f=g-h;
```



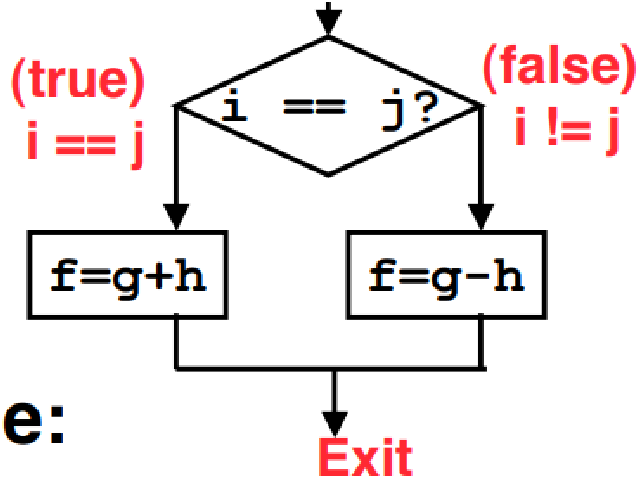
- **Use this mapping:**

```
f: $s0
g: $s1
h: $s2
i: $s3
j: $s4
```

Compiling C if into RISC-V (1/2)

- **Compile by hand**

```
if (i == j) f=g+h;
else f=g-h;
```



- **Final compiled RISC-V code:**

```
        beq $s3,$s4,True   # branch i==j
        sub $s0,$s1,$s2    # f=g-h(false)
        j   Fin            # goto Fin
True:   add $s0,$s1,$s2    # f=g+h (true)
Fin:
```

Summary

- A decision allows us to decide what to execute at run-time rather compile-time
- C decision are made using conditional statements within if, while, do while, for
- RISC-V decision making instructions are conditional branches: beq and bne

Outline

- Branch instructions
- Bitwise/Logical instructions
- Loops in assembly
- Inequality in RISC-V
- Functional calling

Bitwise Operations

- New Perspective: View register as 32 raw bits rather than as a single 32-bit number
- Registers are composed of 32 bits
- We many want to access individual bits (or groups of bits) rather than the whole
- Introduce two new classes of instructions:
  - Logical & Shift Ops

Logical Operators (1/3)

- Two basic logical operators
  - AND: outputs 1 only if both inputs are 1
  - OR: outputs 1 if at least one input is 1
- Truth Table

A	B	A AND B	A OR B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Logical Operators (2/3)

- Logical Instruction Syntax
  - 1   2, 3, 4
  - 1) operation name
  - 2) register that will receive value
  - 3) first operand (register)
  - 4) second operand (register) or immediate constant
- Instruction Names:
  - and, or: Both of these expect the third argument to be a register
  - andi, ori: Both of these expect the third argument to be an immediate

Logical Operators (3/3)

- Note: a->s1, b->s2, c->s3

Instruction	C	RISCV
And	`a = b & c;`	`and  s1,s2,s3`
And Immediate	`a = b & 0x1;`	`andi s1,s2,0x1`
Or	`a = b | c;`	`or   s1,s2,s3`
Or Immediate	`a = b | 0x5;`	`ori  s1,s2,0x5`
Exclusive Or	`a = b ^ c;`	`xor  s1,s2,s3`
Exclusive Or Immediate	`a = b ^ 0xF;`	`xori s1,s2,0xF`

Uses for Logical Operators (1/2)

- anding a bit with 0 produces a 0 at the output
- anding a bit with 1 produces the original bit
- This can be used to create a mask (andi $t0, $t0, 0xFF)

```
       1011 0110 1010 0100 0011 1101 | 1001 1010
mask:  0000 0000 0000 0000 0000 0000 | 1111 1111

The result of anding this:
       0000 0000 0000 0000 0000 0000 | 1001 1010
```

Mask the last 8 bits

Uses for Logical Operators (2/2)

- **“oring”** a bit with 1 produces a 1 at the output
- **“anding”** a bit with 0 produces the original bit
- This can be used to force certain bits of a string to 1s
  - For example, if $t0 contains 0x12345678, then after this instruction
    ori $t0, $t0, 0xFFFF
  - …$t0 contains 0x1234FFFF
    - E.g. the high-order 16-bits are untouched, while the low-order 16 bits are forced to 1s

Shift Instruction (1/4)

- Move (shift) all the bits in a word to the left or right by a number of bits

- **Example: shift right by 8 bits**

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- **Example: shift left by 8 bits**

0001 0010 0011 0100 0101 0110 0111 1000

0011 0100 0101 0110 0111 1000 0000 0000

Shift Instruction (2/4)

- Shift right arith by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- **Example: shift right arith by 8 bits**

1001 0010 0011 0100 0101 0110 0111 1000

1111 1111 1001 0010 0011 0100 0101 0110

Shift Instruction (3/4)

- Shift Instruction Syntax:
  - 1 2, 3, 4
  - 1) operation name
  - 2) register that will receive value
  - 3) first operand (register)
  - 4) shift amount (constant < 32)

```
a *= 8; (in C)
would compile to:
sll    $s0,$s0,3
```

- Since shifting may be faster than multiplication, a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction

Shift Instruction (4/4)

- sra (shift right arithmetic): Shifts right and fills emptied bits by sign extending

Instruction Name	RISC-V
Shift Left Logical	sll s1,s2,s3
Shift Left Logical Imm	slli s1,s2,imm
Shift Right Logical	srl s1,s2,s3
Shift Right Logical Imm	srli s1,s2,imm
Shift Right Arithmetic	sra s1,s2,s3
Shift Right Arithmetic Imm	srai s1,s2,imm

Outline

- Branch instructions
- Bitwise/Logical instructions
- Loops in assembly
- Inequality in RISC-V
- Functional calling

Loops in C/Assembly (1/3)

- Simple loop in C; A[ ] is an array of ints

```
do {
        g = g + A[i];
        i = i + j;
 } while (i != h);
```

- Rewrite this as

```
Loop: g = g + A[i];
      i = i + j;
      if (i != h) goto Loop;
```

```
 g,    h,    i,    j, base of A
$s1, $s2, $s3, $s4, $s5
```

Loops in C/Assembly (2/3)

- Final compiled RISC-V code:

```
Loop: sll $t1,$s3,2    #$t1= 4*i
      add $t1,$t1,$s5  #$t1=addr A
      lw  $t1,0($t1)   #$t1=A[i]
      add $s1,$s1,$t1  #g=g+A[i]
      add $s3,$s3,$s4  #i=i+j
      bne $s3,$s2,Loop# goto Loop
                       # if i!=h
```

- Original code

```
Loop: g = g + A[i];
      i = i + j;
      if (i != h) goto Loop;
```

Loops in C/Assembly (3/3)

- There are three types of loops in C
  - While
  - do … while
  - for
- Each can be rewritten as either of the other two, so the method used in the previous example can be applied to “while” and “for” loops as well

Outline

- Branch instructions
- Bitwise/Logical instructions
- Loops in assembly
- Inequality in RISC-V
- Functional calling

Inequalities in RISC-V (1/2)

- General programs need to test “<” and “>” as well
- Create a RISC-V Inequality instruction
  - Set on Less Than
  - Syntax: slt   reg1, reg2, reg3
  - Meaning: reg1 = (reg2 < reg3)

```
if (reg2 < reg3)
     reg1 = 1;
else reg1 = 0;
```

Inequalities in RISC-V (2/2)

- For example
  - if (g < h) goto Less; #g:$s0, h:$s1
- RISC-V code

```
slt $t0,$s0,$s1 # $t0 = 1 if g<h
bne $t0,$0,Less # goto Less
                # if $t0!=0
                # (if (g<h)) Less:
```

  - Branch if $t0 != 0 -> (g < h)
  - Register $0/$x0 always contains the value 0, so “bne” and “beq” often use it for comparison after an “slt” instruction
  - A slt -> bne pair means if (… < …) goto…

Immediates in Inequalities

- There is an immediate version of slt to test against constants: slti

```
    if (g >= 1) goto Loop
---------------------------------
Loop: . . .

slti $t0,$s0,1      # $t0 = 1 if
                    # $s0<1 (g<1)
beq  $t0,$0,Loop    # goto Loop
                    # if $t0==0
                    # (if (g>=1))
```

- A slti -> beq pair means if (… ≥ …) goto…

What about unsigned numbers?

- **Unsigned** inequality instructions: sltu, sltiu
  - Which sets results to 1 or 0 depending on unsigned comparisons

- **What is value of `$t0`, `$t1`?**

($s0 = FFFF FFFA$_{hex}$, $s1 = 0000 FFFA$_{hex}$)

```
slt $t0, $s0, $s1
sltu $t1, $s0, $s1
```

RISC-V Signed vs. Unsigned

- RISC-V Signed vs. Unsigned is an “overloaded” term
  - **Do/Don’t sign extend**
    (lb, lbu)
  - **Don’t overflow**
    (addu, addiu, subu, multu, divu)
  - **Do signed/unsigned compare**
    (slt, slti/sltu, sltiu)

Summary

- To help the conditional branches
- Make decision concerning inequalities
- We introduce a single instruction
  "Set on Less Than" called slt, slti, sltu, sltiu

Takeaway Questions

C Code:

```
if(i<j) {
   a = b   /* then */
} else {
   a = -b /* else */
}
```

In English:

- If TRUE, execute the <u>THEN</u> block
- If FALSE, execute the <u>ELSE</u> block

RISCV (???):

```
# i→s0, j→s1
# a→s2, b→s3

slt t0 s0 s1
???  t0,??? else
then:
add s2, s3, x0
j    end
else:
sub s2, x0, s3
end:
```

What is ???

Takeaway Questions

- What C code properly fills in the following blank?
  - (A) j >= 2 && j < i
  - (B) j < 2 || j < i
  - (C) j < 2 && j >=i

```
do {i--; } while((z = _______________));
```

```
Loop:
addi  s0,s0,-1
slti  t0,s1,2
bne   t0,x0 Loop
slt   t0,s1,s0
bne   t0,x0 ,Loop
```

Takeaway Questions

- What C code properly fills in the following blank?
    - (A) j >= 2 && j < i
    - (B) j < 2 || j < i
    - (C) j < 2 && j >=i

```
do {i--; } while((z = ______________));
```

```
Loop:                   # i→s0, j→s1
addi   s0,s0,-1         # i = i - 1
slti   t0,s1,2          # t0 = (j < 2)
bne    t0,x0 Loop       # goto Loop if t0!=0
slt    t0,s1,s0         # t1 = (j < i)
bne    t0,x0 ,Loop      # goto Loop if t0!=0
```

Outline

- Branch instructions
- Bitwise/Logical instructions
- Loops in assembly
- Inequality in RISC-V
- Functional calling

Calling Conventions

- Calle**R**: the calling function
- Calle**E**: the function being called
    - E.g. Alice is caller and Bob is callee
- What instructions can accomplish the functional call?

```
void Alice () {
    Bob ();
}
```

Function Call Bookkeeping

- Registers play a major role in keeping track of information for function call
- Register conventions
  - **Return address** $ra
  - **Arguments** $a0 - $a7
  - **Return value** $s0 - $s1
  - **Local variables** $t0 - $t6
- The stack is also used; more later

x0	zero	zero
x1	ra	return address
x2	sp	stack pointer
x3	gp	global data pointer
x4	tp	thread pointer
x5	t0	**temps (caller save)**
x6	t1	
x7	t2	
x8	s0/fp	frame pointer
x9	s1	**saved (callee save)**
x10	a0	function args or return values
x11	a1	
x12	a2	function arguments
x13	a3	
x14	a4	
x15	a5	function arguments
x16	a6	
x17	a7	
x18	s2	**saved (callee save)**
x19	s3	
x20	s4	
x21	s5	
x22	s6	
x23	s7	
x24	s7	
x25	s9	
x26	s10	
x27	s11	
x28	t3	**temps (caller save)**
x29	t4	
x30	t5	
x31	t6	

Instruction Support for Functions (1/6)

C code

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

RISC-V

address
1000
1004
1008
1012
1016

2000
2004

In RISC-V 32, all instructions are 4 bytes (32-bits), and stored in memory just like data. Here we show the addresses of where the programs are stored.

Instruction Support for Functions (2/6)

C code

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

- Jump (j)
  - j label
- Jump Register (jr)
  - jr src

RISC-V

```
address
1000    add     $a0, $s0, $x0      # x = a
1004    add     $a1, $s1, $x0      # y = b
1008    addi    $ra, $x0, 1016     # $ra = 1016
1012    j       sum                # jump to sum
1016    …
2000    sum:    add  $t0, $a0, $a1
2004    jr      $ra
```

Instruction Support for Functions (3/6)

C code

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

Question: Why use jr here? Why not simply use j?

Answer: "sum" might be called by many functions, so we can't return to a fixed place. The calling proc to "sum" must be able to say "return here".

RISC-V

```
...
2000    sum:    add   $t0, $a0, $a1
2004    jr            $ra
```

Instruction Support for Functions (4/6)

- Jump and link (jal)
  - Single instruction to jump and save the return address

Before:

1008 add $ra, $x0 $1016 # $ra = 1016
1012 j sum # goto sum

After:

1008 jal sum

Why have a jal?
jal moves a new value into the PC and simultaneously saves the old value in register x1 ($ra) can get back from the subroutine to the instruction immediately following the jump by transferring control back to PC in register x1

Instruction Support for Functions (5/6)

- Jump and link (jal)
  - jal label
  - Behaves like the simple jump instruction (j), but also stores a return address in register 31 ($ra)
  - Step 1 (link):
    - Save the address of next instruction into $ra
    - The next instruction (PC + 4)
    - Why the next instruction? Why not the current one?
  - Step 2 (jump)
    - Jump to the given label

Instruction Support for Functions (6/6)

- Jump Register (jr)
  - jr src
  - Instead of providing a label to jump to, the jr instruction provides a register that contains an address to jump to
  - Only useful if we know the exact address to jump to
  - Very useful for function calls:
    - jal stores return address in register
    - jr $ra jumps back to that address

Nested Procedures (1/2)

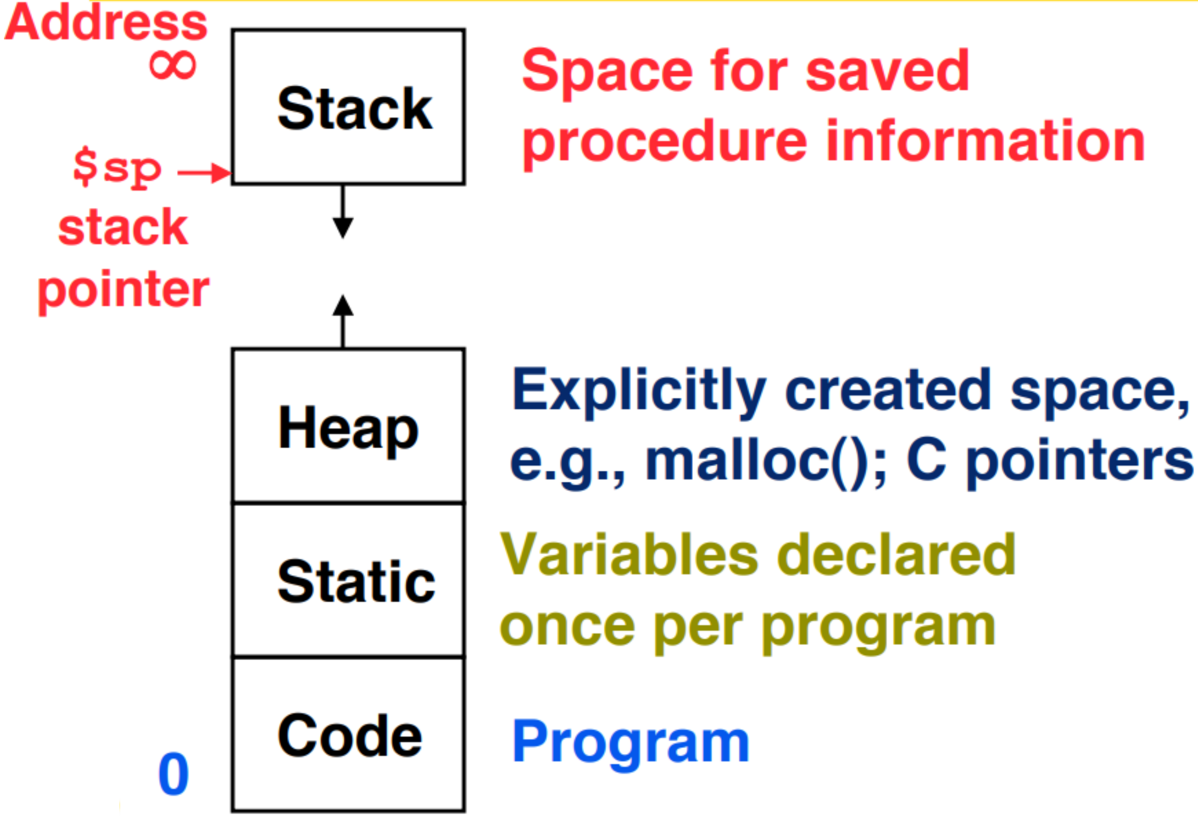
- **sumSquare** nested procedure
  - sumSquare is calling mult
  - There is a value in $ra that sumSquare wants to jump back to, but the call to mult will overwrite this
  - Need to save sumSquare return address before call to mult

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

Nested Procedures (2/2)

- In general, you may need to save some other info in addition to $ra
- When a C program is run, there are 3 important memory area memory areas allocated
  - **Static:** Variables declared once per program, cease to exist only after execution completes. E.g. C globals
  - **Heap:** Variables declared dynamically. E.g. malloc()
  - **Stack:** Space to be used by procedure during execution; this is where we can save register values

C Memory Allocation Review



Using the Stack (1/2)

- We have a register $sp which always points to the last used space in the stack
- To use the stack, we decrement this pointer by the amount of space we need and then fill it with info.
- How do we compile this?

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

Using the Stack (2/2)

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

```
sumSquare:
"push"  addi  $sp,  $sp,     -8    #space on stack
        sw    $ra,  4($sp)         #save ret addr
        sw    $a1,  0($sp)         #save y

        add   $a1,  $a0,     $x0   # mult(x, x)
        jal   mult                 # call mult

        lw    $a1,  0($sp)         # restore y
        add   $a0,  $a0,     $a1   # mult() + y
"pop"   lw    $ra,  4$($sp)        # get ret addr
        addi  $sp   $sp,     8     # restore stack
        jr    $ra
mult: …
```

Steps for Making a Procedure Call

- (1) Save necessary values onto the stack
- (2) Assign argument(s), if any
- (3) jal call
- (4) Restore values from stack

Rules for Procedures

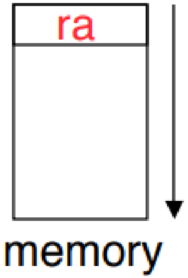
- Call with a jal instruction, returns with a jr $ra
- Accepts up to 8 arguments in $a0 - $a7
  - Any more arguments should be passed on the stack
- Return value is always in $s0 (and if necessary in $s1)

Basic Structure of a Function

Prologue

```
entry_label:
addi $sp,$sp, -framesize
sw $ra, framesize-4($sp)   # save $ra
save other regs if need be
```

Body · · · **(call other functions…)**

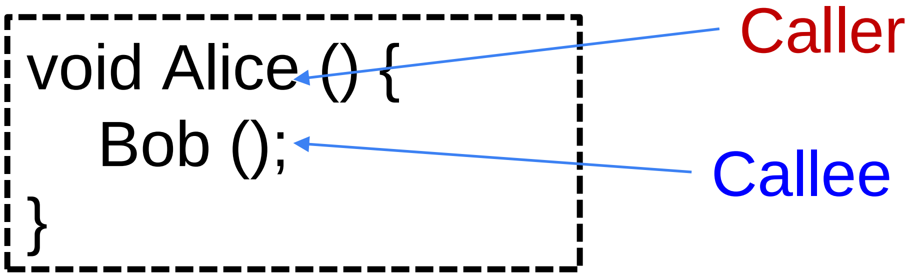


Epilogue

```
restore other regs if need be
lw $ra, framesize-4($sp)   # restore $ra
addi $sp,$sp, framesize
jr $ra
```

Register Conventions (1/3)

- When callee returns from executing, the caller needs to know which register may have changed and which are guarantee to be unchanged
- Register conventions
  - A set of generally accepted rules as to which registers will be unchanged after a procedure call (jal) and which may be changed



Register Conventions (2/3) -- Saved

- $**x0:** No Change. Always 0
- **$s0 - $s7**: Restore if you change
  - That’s why they’re called save registers. If the callee changes these in any way, it must restore the original values before returning
- **$sp:** Restore if you change
  - The stack pointer must point to the same place before and after the “jal” call, or else the caller won’t be able to restore values from the stack
- **All saved register start with** S!

Register Conventions (3/3) -- Volatile

- **$ra:** Can Change
  - The jal call itself will change this register. Caller needs to save on the stack if nested call
- **$a0 - $a1**: Can Change
  - These will contain the new return values
- **$t0 - $t6:** Can Change
  - That’s why they’re called temporary; any procedure may change them at any time. Caller needs to save if they will need them afterwards

Summary

- Functions called with jal, return with jr $ra
- Use the stack to save anything you need. Just be sure to leave it the way you found it
- Instructions we know so far
  - **Arithmetic**: add, addi, sub, addu, addiu, subu
  - **Memory**: lw, sw
  - **Decision**, beq, bne, slt, slti, sltu, sltiu
  - **Unconditional branches** (Jumps): j, jal, jr

Summary

- Registers we know so far

The Constant 0	**$x0**	**$zero**
Return address	**$x1**	**$ra**
Stack pointer	**$x2**	**$sp**
Global data pointer	**$x3**	**$gp**
Thread pointer	**$x4**	**$tp**
Temporary	**$x5-$x7**	**$t0-t2**
Frame pointer	**$x8**	**$s0/$fp**
Saved	**$x9**	**$s1**

Summary

- Registers we know so far

Return values/Arguments	**$x10-$x11**	**$a0-$a1**
Function Arguments	**$x12-x17**	**$a2-a7**
Saved	**$x18-x27**	**$s2-$s11**
Temporary	**$x28-x31**	**$t3-$t6**