Given the following function header,
int foo(int a, int b);
what will be on the stack before any of the calculations in foo are performed? Assume foo() calls some other function.
What will be on the stack on a call to `int foo(int a, int b, int c, int d, int e, int f)`?
DIFFERENT BASES
Number Systems

- Decimal
  \[ \sum_{i} d_i \times 10^i \]
- Binary
  \[ \sum_{i} b_i \times 2^i \]
- Hexadecimal
  \[ \sum_{i} h_i \times 16^i \]
Converting From Decimal to Binary

```java
string s = "";
while(value > 0){
    digit = val % 2;
    val = val / 2;
    s = digit + s;
}
```
Converting From Decimal to Binary

- What is the largest power of 2 that fits into the decimal number?
  - That binary digit will be 1
- Subtract off that value from the number
- Repeat until number is 0
Converting Binary To Hexadecimal

- Hexadecimal
  - 0 through 9, A through F
  - 0000 through 1001, 1010 through 1111
- Start on the right. For every 4 binary digits, convert to single hexadecimal digit.
  
  (The reverse process works too!)
Assembler
# assign.asm

# simple program to modify a global variable

.data  # add what follows to the data section of the load mod.
   x: .word 5

.text  # add what follows to the text section of the load mod.
   .align 2  # Align on word boundaries
   .globl main  # "exports" the symbol main so it is accessible to other modules

main:
   # we don't need a frame
   la  $t0, x  # $t0 = &x
   lw  $t1, ($t0)  # $t1 = x
   addi $t1,$t1,2  # $t1 = $t1 + 2
   sw  $t1, ($t0)  # x = $t1
   jr  $ra  # return - main is a function, too
Assembly File

- Segments
  - .data
    - Integer (.word), character (.byte), arrays of these
  - .text
    - Instructions
    - main should be first instruction and needs to be specified as .globl
Two Pass Assembler

- **Pass 1**
  - Locates all labels
  - Determines address given to each label
  - Checks syntax

- **Pass 2**
  - Generates binary encoding of data and instructions, replacing labels with corresponding addresses
Pass One

- At each line in file,
  - Detect any syntax errors
  - Determine how many bytes need to be allocated to handle that line
  - On encountering a label
    - Put into SymbolTable
    - If on left followed by a colon, set the address of the label in the symbol table
      - Start of segment + bytes already seen
Pass Two

- Assuming all labels defined and no syntax errors in Pass One.
  - For each line,
    - Generate binary encoding
    - If instruction contains label as an operand, use binary encoding of label’s address taken from SymbolTable
.data

x: .word 5

.text

.align 2

.globl main

main:
   la  $t0, x           # $t0 = &x
   lw  $t1, ($t0)       # $t1 = x
   addi $t2, $r0, 5     # $t2 = 5

loop:  beq  $t2, $r0, done    # if($t2 == 0) go to done
       addi $t1,$t1,2    # $t1 = $t1 + 2
       addi $t2, $t2, -1 # $t2 = $t2 - 1
       b      loop

done:  sw  $t1, ($t0)       # x = $t1
         jr  $ra
x: 0x10010000
main: 0x400000
loop: 0x400010
done: 0x400020
.data
str: .asciiz "Hey!"
x: .word 7 3
y: .word -1
.text
.align 2
.globl main
main:
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    la $a0,str
    jal print
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra
print: addi $sp, $sp, -4
    sw $ra, 0($sp)
    li $v0, 4
    syscall
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra
str: 0x10010000
x: 0x10010008
y: 0x10010010
main: 0x400000
print: 0x400020
Creating Executables
Object Files

Created by assembler from assembly language program that contains machine language instructions, data, and info for placing instructions in memory.

- Object file header
  - Describes size and position of other segments of object file

- Text segment
  - Machine language code

- Data segment
  - Static and dynamic data

- Relocation information
  - Identifies instructions and data words that depend on absolute addresses when program loaded into memory
  - Ex. reloc may specify instruction as “sw $t0, Y” with dependency on address of data Y
Object Files (cont)

- Symbol table
  - Labels that are not defined, such as external references
  - E.g., symbol table might contain Y and address (unknown) of labels

- Debugging information
  - Includes info that associates machine instructions with source code
Object File Header

- object file header
- text
- data
- relocation
- symbol table
Executable File

Created by linker that stitches object files together and can be run on hardware.

- Places code and data modules symbolically in memory
- Determines addresses of data and instruction labels
- Patches both internal and external references
<table>
<thead>
<tr>
<th>executable file header</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
</tr>
<tr>
<td>data</td>
</tr>
</tbody>
</table>
Representing Instructions in Machine Language

- **op**: Basic operation (opcode)
- **rs**: First source register
- **rt**: Second source register
- **rd**: Destination register
- **shamt**: Shift amount
- **funct**: Function
Representing Instructions in Machine Language

R-Type (register) – opcode 000000

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6b</td>
<td>5b</td>
<td>5b</td>
<td>5b</td>
<td>5b</td>
<td>6b</td>
</tr>
</tbody>
</table>

I-Type (data transfer)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6b</td>
<td>5b</td>
<td>5b</td>
<td>16b</td>
</tr>
</tbody>
</table>

J-Type (jump) – opcode 00001x  
Note: NOT branch

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6b</td>
<td>26b</td>
</tr>
</tbody>
</table>
Note:

• J-Type instructions are only j and jal
• I-Type is all instructions immediate operand, branch instructions, and load and store instructions
• R-Type is all arithmetic and logic with all operands in registers, shift instructions, and register direct jump instructions (jalr and jr)
### MIPS Addressing Modes

1. **Immediate addressing**

   | op | rs | rt | Immediate |

   Immediate: holds a constant

2. **Register addressing**

   | op | rs | rt | rd | ... | funct |

   Register addressing: a register number specifies which register holds the data.

3. **Base addressing/Register indirect**

   | op | rs | rt | Address |

   Base addressing/Register indirect: Offset added to contents of register to determine address of memory location for data.

---

**Registers**

**Memory**

- Byte
- Halfword
- Word
MIPS Addressing Modes

Take the value encoded in the instruction, shift two to left to make that a word aligned address (instructions are all 4B), then add to nextPC (branches)

4. PC-relative addressing

Take the value encoded in the instruction, shift two to left to make that a word aligned address, concatenate with first 4 bits of nextPC (Jump/Jal)
Example Encoding

0000 0000 1010 1111 1000 0000 0010 0000
### op(31:28)

<table>
<thead>
<tr>
<th>28–26</th>
<th>31–29</th>
<th>0(000)</th>
<th>1(001)</th>
<th>2(010)</th>
<th>3(011)</th>
<th>4(100)</th>
<th>5(101)</th>
<th>6(110)</th>
<th>7(111)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0(000)</td>
<td>R-format</td>
<td>Bltz/gez</td>
<td>jump</td>
<td>jump &amp; link</td>
<td>branch eq</td>
<td>branch ne</td>
<td>blez</td>
<td>bgtz</td>
<td></td>
</tr>
<tr>
<td>1(001)</td>
<td>add immediate</td>
<td>addiu</td>
<td>set less than imm.</td>
<td>set less than imm. unsigned</td>
<td>andi</td>
<td>ori</td>
<td>xor1</td>
<td>load upper immediate</td>
<td></td>
</tr>
<tr>
<td>2(010)</td>
<td>TLB</td>
<td>FlPt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3(011)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4(100)</td>
<td>load byte</td>
<td>load half</td>
<td>lw1</td>
<td>load word</td>
<td>load byte unsigned</td>
<td>load half unsigned</td>
<td>lwr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5(101)</td>
<td>store byte</td>
<td>store half</td>
<td>swl</td>
<td>store word</td>
<td>swr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6(110)</td>
<td>load linked word</td>
<td>lwcl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7(111)</td>
<td>store cond. word</td>
<td>swcl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### op(31:26)=010000 (TLB), rs(25:21)

<table>
<thead>
<tr>
<th>23–21</th>
<th>25–24</th>
<th>0(000)</th>
<th>1(001)</th>
<th>2(010)</th>
<th>3(011)</th>
<th>4(100)</th>
<th>5(101)</th>
<th>6(110)</th>
<th>7(111)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0(000)</td>
<td>mfc0</td>
<td>cfc0</td>
<td>mtc0</td>
<td>ctc0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1(01)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2(10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3(11)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### op(31:26)=000000 (R-format), funct(5:0)

<table>
<thead>
<tr>
<th>2–0</th>
<th>5–3</th>
<th>0(000)</th>
<th>1(001)</th>
<th>2(010)</th>
<th>3(011)</th>
<th>4(100)</th>
<th>5(101)</th>
<th>6(110)</th>
<th>7(111)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0(000)</td>
<td>shift left logical</td>
<td>shift right logical</td>
<td>sra</td>
<td>sllv</td>
<td>srlv</td>
<td>srav</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1(001)</td>
<td>jump register</td>
<td>jalr</td>
<td>syscall</td>
<td>break</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2(010)</td>
<td>mfhi</td>
<td>mthi</td>
<td>mfo</td>
<td>mtlo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3(011)</td>
<td>mult</td>
<td>multu</td>
<td>div</td>
<td>divu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4(100)</td>
<td>add</td>
<td>addu</td>
<td>subtract</td>
<td>subu</td>
<td>and</td>
<td>or</td>
<td>xor</td>
<td>not or (nor)</td>
<td></td>
</tr>
<tr>
<td>5(101)</td>
<td>set l.t.</td>
<td>set l.t. unsigned</td>
<td>set l.t. signed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6(110)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7(111)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Fields</td>
<td>Comments</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>-----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field size</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td>All MIPS instructions are 32 bits long</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-format</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td>Arithmetic instruction format</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I-format</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td>address/immediate</td>
<td>Transfer, branch, imm. format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J-format</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td>target address</td>
<td>Jump instruction format</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Branch Instructions

I-Type (data transfer)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6b</td>
<td>5b</td>
<td>5b</td>
<td>16b</td>
</tr>
</tbody>
</table>

- Computing target address:
  - Take 16b address
  - Shift 2b to left
  - Add result to PC+4
Jump Instructions

J-Type (jump) – opcode 00001x

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6b</td>
<td>26b</td>
</tr>
</tbody>
</table>

- Computing target address:
  - Take 26b address
  - Shift 2b to left
  - Append 28b to first 4b of PC
.data
x: .word 5
str:.asciiz "A"
y: .word -1

.text
.align 2  # Align on word boundaries
.globl main
main:
    la       $t1, x
    lw       $t0, 0($t1)
foo:addi $t0,$t0,2
    sw       $t0, 0($t1)
b       foo
    jr       $ra