Run time environment of a MIPS program

- Stack pointer
- Temporary local variables
- Return address
- Saved argument registers beyond a0-a3

Frame pointer

Low address

Growth of stack

High address
A translation hierarchy

HLL program

COMPILER

Assembly language program

ASSEMBLER

Machine language module

LINKER ← Library routine

Executable machine language program

LOADER

Memory
What are Assembler directives?

Instructions that are not executed, but they tell the assembler about how to interpret something. Here are some examples:

```
.text

{Program instructions here}

.data

{Data begins here}

.byte 84, 104, 101

.ascii "The quick brown fox"

.float f1, . . . , fn

.word w1, . . . , wn
```
How does an assembler work?

In a two-pass assembler

PASS 1: Symbol table generation
PASS 2: Code generation

The operation of a two-pass assembler has been briefly explained in the class on 2/9/10.
Other architectures

Not all processors are like MIPS.

Example. Accumulator-based machines

A single register, called the accumulator, stores the operand before the operation, and stores the result after the operation.

<table>
<thead>
<tr>
<th>Load</th>
<th>x</th>
<th># into acc from memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>y</td>
<td># add y from memory to the acc</td>
</tr>
<tr>
<td>Store</td>
<td>z</td>
<td># store acc to memory as z</td>
</tr>
</tbody>
</table>

Can we have an instruction

```
add z, x, y  # z:= x + y, (x, y, z in memory)
```

For some machines, YES, not in MIPS
**Load–store machines**

MIPS is a load–store architecture. Only load and store instructions access the memory, all other instructions use registers as operands. What is the motivation?

*Register access is much faster than memory access, so the program will run faster.*

**Reduced Instruction Set Computers (RISC)**

- The instruction set has only a small number of frequently used instructions. This lowers processor cost, without much impact on performance.
- All instructions have the same length.
- Load–store architecture.

Non-RISC machines are called CISC (Complex Instruction Set Computer). Example: Pentium
Another classification

3-address  add  r1, r2, r3  (r1 ← r2 + r3)
2-address  add  r1, r2  (r1 ← r1 + r2)
1-address  add  r1  (to the accumulator)
0-address or stack machines  (see below)

Example of stack architecture

Push x
Push y
Push z
Add
Multiply
Pop z

Computes z = x * (y + z)
Computer Arithmetic

How to represent negative integers? The most widely used convention is 2’s complement representation.

<table>
<thead>
<tr>
<th>+14</th>
<th>0, 1 1 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-14</td>
<td>1, 0 0 1 0</td>
</tr>
</tbody>
</table>

Largest integer represented using n-bits is $+2^{n-1} - 1$

Smallest integer represented using n-bits is $-2^{n-1}$

Review binary-to decimal and binary-to-hex conversions.
Review BCD (Binary Coded Decimal) and ASCII codes.
How to represent fractions?
Overflow

+12 = 0, 1 1 0 0  
+12 = 0, 1 1 0 0  
+2  = 0, 0 0 1 0  
+7  = 0, 0 1 1 1  
add    
+14 = 0, 1 1 1 0  
?    = 1, 0 0 1 1  
add

Addition of a positive and a negative number does not lead to overflow. How to detect overflow?
The following sequence of MIPS instructions can detect overflow in signed addition of \( t1 \) and \( t2 \):

\begin{verbatim}
addu \( t0, t1, t2 \) # add unsigned
xor \( t3, t1, t2 \) # check if signs differ
slt \( t3, t3, zero \) # \( t3=1 \) if signs differ
bne \( t3, zero, no_overflow \)
xor \( t3, t0, t1 \) # sum sign = operand sign?
slt \( t3, t3, zero \) # if not, then \( t3=1 \)
bne \( t3, zero, overflow \)
no_overflow:
...
...
overflow:
<Do something to handle overflow>
\end{verbatim}
The Basics of Exception Handling

MIPS uses two coprocessors: C0 and C1 for additional help. C0 primarily helps with exception handling, and C1 helps with floating point arithmetic.

Interrupts

Initiated outside the instruction stream
Arrive asynchronously (at no specific time),
Example:
  o  I/O device status change
  o  I/O device error condition

Traps

Occur due to something in instruction stream.
Examples:
  o  Unaligned address error
  o  Arithmetic overflow
  o  System call
**MIPS coprocessor** CO has a *cause register* (Register 13) that contains a 5-bit code to identify the cause of an exception.

<table>
<thead>
<tr>
<th>Cause register</th>
</tr>
</thead>
<tbody>
<tr>
<td>pending interrupt</td>
</tr>
<tr>
<td>15-10</td>
</tr>
</tbody>
</table>

MIPS instructions that cause overflow (or some other violation) lead to an *exception*, which sets the *exception code*. It then switches to the *kernel mode* (designated by a bit in the *status register* of CO, register 12) and transfer control to a predefined address to invoke a routine (*exception handler*) for handling the exception.

L: \[ \text{add } $t0, $t1, $t2} \quad \text{overflow!} \\
\text{Return address (L+4) is saved in EPC} \\
\text{Next instruction}

Exception handler routine:
\[ \text{ra } \leftarrow \text{EPC} \]
\[ \text{jr } ra \]

Exceptions cause *unscheduled procedure calls*. 

- **MIPS coprocessor**: A coprocessor for the MIPS architecture, often used to enhance processing power. 
- **Cause register**: A register in MIPS that holds a 5-bit code to identify the cause of an exception. 
- **Exception**: A condition that disrupts the normal execution of a computer program. 
- **Kernel mode**: The mode in which the operating system has full control over the computer's resources. 
- **Status register**: A register in MIPS that contains bits that control the system's operation. 
- **Exception handler**: A routine that handles exceptions when they occur. 
- **EPC**: Exception Program Counter. 
- **Jr**: Jump Register, used to transfer control to another location in the program. 
- **Unscheduled procedure calls**: Calls to routines that are not part of the usual program flow.
More Programming Examples

Copying a string

Each char is represented by an ASCII byte. The string is terminated by a **Null** in ASCII). Reg $s0 will hold the array index.

```assembly
add $s0, $zero, $zero    # i = 0
L1: add $t1, $a1, $s0    # address of y[i] in t1
    lb $t2, 0($t1)      # t2 = y[i]
    add $t3, $a0, $s0   # address of x[i] in t3
    sb $t2, 0($t3)      # x[i] = y[i]
    addi $s0, $s0, 1    # i = i+1
    bne $t2,$zero, L1   # if y[i]≠0 then goto L
```

load byte