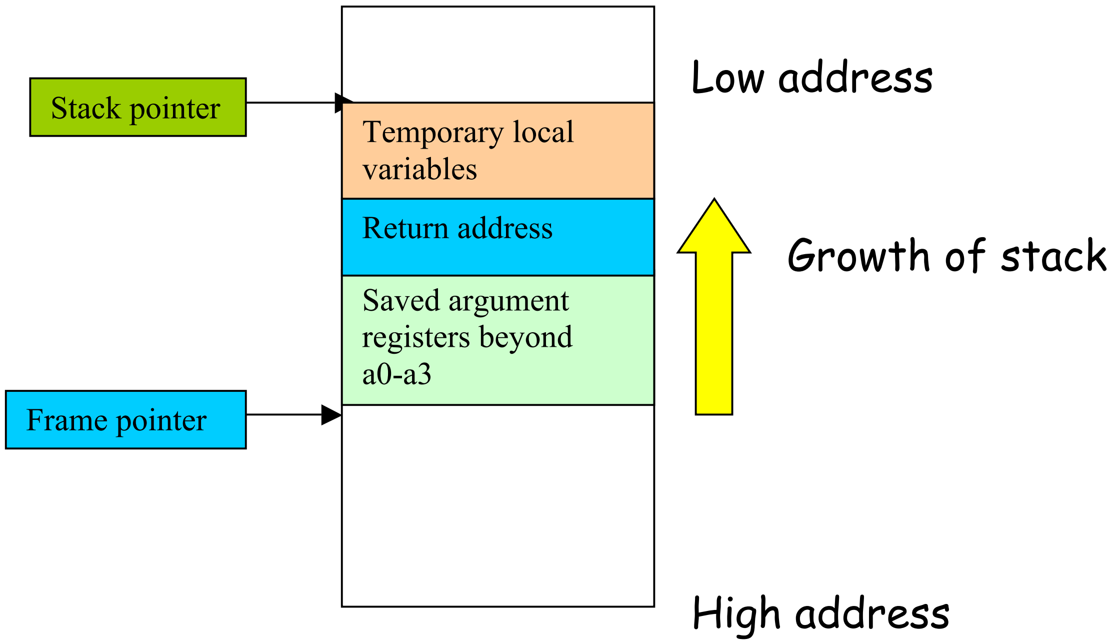
Run time environment of a MIPS program



A translation hierarchy

$$\begin{tikzcd}
\text{HLL program} \arrow[d] & \\
\text{COMPILER} \arrow[d] & \\
\text{Assembly language program} \arrow[d] & \\
\text{ASSEMBLER} \arrow[d] & \\
\text{Machine language module} \arrow[d] & \\
\text{LINKER} \arrow[d] & \text{Library routine} \arrow[l] \\
\text{Executable machine language program} \arrow[d] & \\
\text{LOADER} \arrow[d] & \\
\text{Memory} &
\end{tikzcd}$$

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
. asciiz "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

To be explained in the class …

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.

```
Load     x        # into acc from memory
Add      y        # add y from memory to the acc
Store    z        # store acc to memory as z
```

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

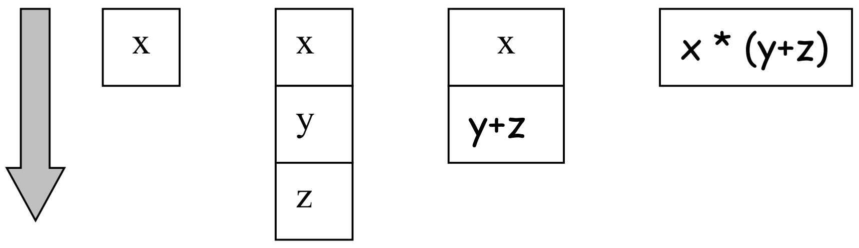
Push y

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.

+14	=	0, 1 1 1 0
-14	=	1, 0 0 1 0

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

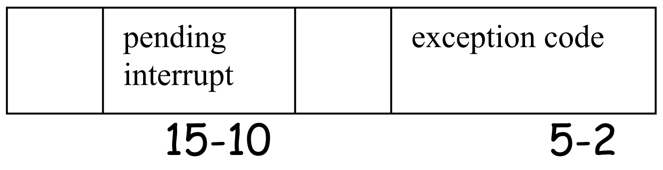
+12	=	0, 1 1 0 0	+12	=	0, 1 1 0 0	
+2	=	0, 0 0 1 0	+7	=	0, 0 1 1 1	
add						add
+14	=	0, 1 1 1 0	?	=	1, 0 0 1 1	

Addition of a positive and a negative number does not lead to overflow. How to detect overflow?

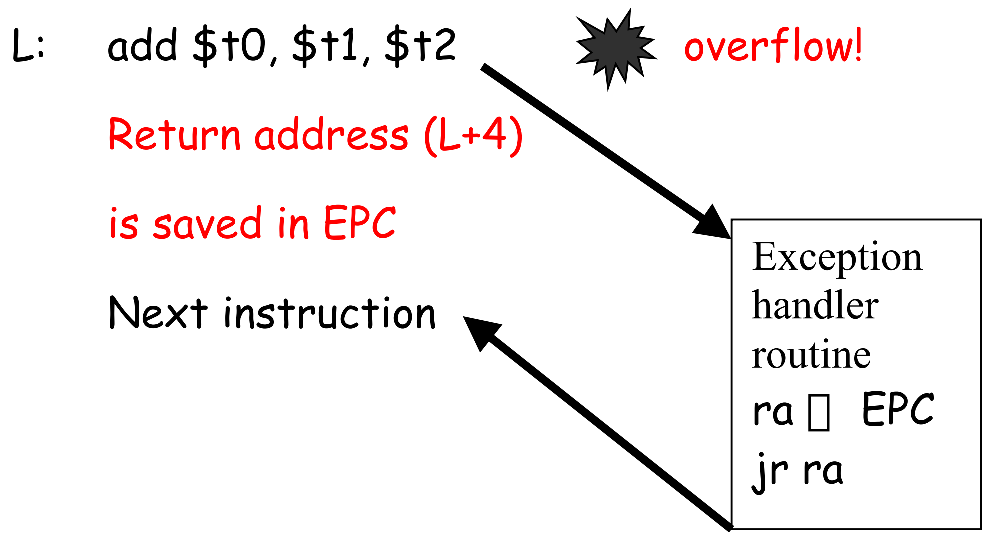
Exceptions

MIPS coprocessor has a cause register that contains a 4-bit code to identify the cause of an exception

Cause register



MIPS instructions that cause overflow (or some other violation) lead to an ***exception*** (also called an ***interrupt***), and transfer control to a predefined address to invoke a routine (exception handler) for handling the exception.



Exceptions cause unscheduled procedure calls.

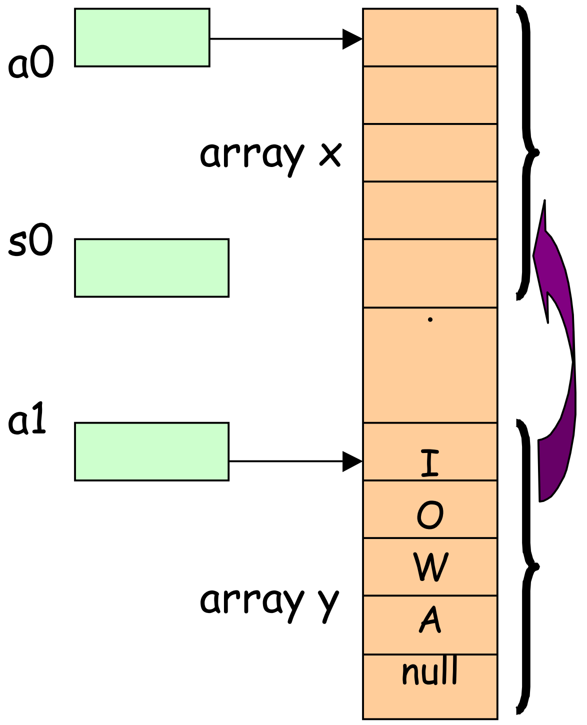
The following sequence of MIPS instructions can detect overflow in signed addition of $t1 and $t2:

```
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>
```

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.



```
      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 L1
```

Load byte