Frame-based Linkage Convention

- This lecture builds further on the stack-based linkage convention to create a frame-based linkage convention. A stack frame is the section of the run-time stack that holds the data of a subroutine.
- What is a local variable in a higher level language?

A local variable is a software entity that holds values for a subroutine while the subroutine is active.
Implementation of Local Variables

• In a high-level language, a local variable is implemented in a location on the run-time stack. Each time a subroutine is activated, new locations for variables are pushed onto the stack. The section of the stack for each activation is called a stack frame or an activation record. The frame pointer holds the address of the stack frame for a subroutine.

• When a subroutine returns to its caller the stack frame is popped from the stack. Thus, local variables only exist as memory locations while a subroutine is active. A subroutine is active if it is currently executing, or if a subroutine it called is active.

• The format of a stack frame used by MIPS language processors is complicated. There are many situations that must be handled and many optimizations. It takes a compiler to do it correctly. This lecture describes a much simplified stack frame.

• The important part is to understand what a local variable is: a location on the run-time stack. This is an important idea in Computer Science, one you will run into repeatedly as you study advanced topics.

Picture of a Stack Frame

• Programming languages also have global variables. Also, an inner program block can use variables that are not its local variables which are defined in a containing block. Let us skip these messy details and implement local variables, only. The details may be covered in a course on compilers or in a course on programming languages.

• The picture shows what a stack frame (in our simplified format) looks like when a subroutine is active. As always for us, each item on the stack is four bytes long.

• As previously, the caller saves the "T" registers that contain values it needs, and the callee (the subroutine) saves the "S" registers it might change. But now room is pushed onto the stack for (in the example) four local variables a, b, i and j.

• While the subroutine is active, the frame pointer, register $30, points to the top of the stack. (Remember, our stacks grow downward, so in the picture $fp is pointing at the last word that was pushed onto the stack, the "top" of the stack.)
Example

• Write the instruction that loads register $t5 with the value held in the variable $a$:

$$\text{lw } $t5,\_\_\_ (\_\_\_)$$

$$\text{lw } $t5,12(\$fp)$$

Frame Pointer

• Register $s30 is reserved, by software convention, for use as a frame pointer. In the extended assembler it has the mnemonic name $fp$. When a subroutine starts running, the frame pointer and the stack pointer contain the same address.

• But the stack (and the stack pointer) may be involved in arithmetic expression evaluation. This often involves pushing and popping values onto and off of the stack. If $sp$ keeps changing, it is hard to write code that accesses a fixed location on the stack, like a variable.

• To make things easy for compilers (and for human assembly language programmers) it is convenient to have a frame pointer that does not change its value while a subroutine is active. The variables will always be the same distance from the unchanging frame pointer.

• When a caller calls a subroutine, the caller's frame pointer is pushed onto the stack along with the other caller-saved registers. Now the subroutine sets the frame pointer to the address of the new stack frame.
Example

- When a subroutine first starts executing, is the address in the frame pointer equal to the address of the top of the stack?

  Yes. However, the stack may grow as the subroutine executes. But the frame pointer for the subroutine does not change. The stack pointer ($sp$) is always equal to the top of stack.

Sample Code

- Imagine that the following statement is part of the subroutine whose stack frame is at the right:

  \[ a = b + i + j; \]

- This is how a compiler might implement that statement:

```assembly
lw $t0,8($fp)    # get b
lw $t1,4($fp)    # get i
lw $t2,0($fp)    # get j
addu $t3,$t0,$t1 # b + i
addu $t3,$t3,$t2 # b + i + j
sw $t3,12($fp)   # a =
```
Frame-based Linkage Convention

A real-world linkage convention allows many types of objects to go into a stack frame. Our rules are much simpler:

**Calling a Subroutine (done by the caller):**
- Push any registers $t0$-$t9$ that contain values that must be saved. Push the registers in numerical order.
- Put argument values into $a0$-$a3$.
- Call the subroutine using `jal`.

**Subroutine Prolog (done by the subroutine):**
- Push $ra$ (always).
- Push the caller's frame pointer $fp$.
- Push any of the registers $s0$-$s7$ that the subroutine might alter.
- Initialize the frame pointer: $fp = sp - \text{space for variables}$. The "space for variables" is four times the number of local variables. (Remember that subtracting from $sp$ grows the stack).
- Initialize the stack pointer: $sp = fp$.

**Subroutine Body:**
- At this point the stack looks like the picture at right.
- The subroutine may alter any "T" or "A" register, or any "S" register that it saved in the prolog.
- The subroutine may push and pop values on the stack using $sp$.
- If the subroutine calls another subroutine, then it does so by following these rules.

**Subroutine Epilog (done at the end of the subroutine):**
- Put return values in $v0$-$v1$
- $sp = fp + \text{space for variables}$.
- Pop into $s0$-$s7$ any values for them that were previously saved in the frame.
- Pop the caller's frame pointer into $fp$.
- Pop $ra$ (always).
- Return to the caller using `jr $ra`.

**Regaining Control from a Subroutine (done by the caller):**
- Pop any registers $t0$-$t9$ that the caller previously pushed.
These rules are complicated. In broad outline it works the same way as the previous lecture's stack-based linkage convention. The additional complications are due to implementing variables as locations on the stack. Here is a picture. It shows the sections of subroutine linkage. The basic tasks of each section are:

- **Subroutine Call**: Push any "T" registers that contain values that are needed. Put arguments in "A" registers. `jal` to the subroutine.
- **Prolog**: Push $ra and the caller's $fp. Push any "S" register the subroutine will alter. Initialize the subroutine's $fp and $sp.
- **Body**: Normal code, except it must follow these conventions if it calls another subroutine. "T" and "A" registers can be used freely, as can any "S" registers that were saved in the prolog.
- **Epilog**: Put return values in "V" registers. Reset $sp. Pop any "S" registers. Pop the caller's $fp and $ra. `jr $ra` back to the caller.
- **Regaining Control**: Pop any previously pushed "T" registers.

Questions

- When the caller gets control back, are its frame pointer and stack pointer the same as when it called the subroutine?

  Yes.

- Is there a limit to how many variables a subroutine may have?

  No (practically).
Example Program

• The number of registers that MIPS (or other processors) has does not limit the number of variables that subroutines can have. As many variables as you want can be allocated on the stack. Here is an example program:

```c
main() {
    int a;
    a = mysub( 6 );
    print( a );
}

int mysub( int arg ) {
    int b,c;
    b = arg*2;
    c = b + 7;
    return c;
}
```

• To the operating system, main() is a subroutine. When main() first gets control it must follow the rules under "subroutine prolog".

```c
main()
```

```
# main() {
#   int a;
#   a = mysub( 6 );
#   print( a );
# }

.text
.globl main
main:
    # prolog
    sub $sp,$sp,4 # 1. Push return address
    sw $ra,($sp)
    sub $sp,$sp,4 # 2. Push caller's frame pointer
    sw $fp,($sp)
    sub ___,____,____ # 3. No S registers to push
    _____ $sp,$fp # 4. $fp = $sp - space_for_variables
    _____ $sp,$fp # 5. $sp = $fp
    # subroutine call
    ....
    # return from subroutine
    ....
    # epilog
    jr $ra # return to OS
```
Subroutine Call

- At this point we have "compiled" into assembly language the first three lines of the "C" program. Next, the program calls the subroutine mysub().

```assembly
.globl main
main:
    # prolog
    sub $sp,$sp,4       # 1. Push return address
    sw $ra,($sp)
    sub $sp,$sp,4       # 2. Push caller's frame pointer
    sw $fp,($sp)
    sub $sp,$fp,4       # 3. No S registers to push
    move $sp,$fp        # 4. $fp = $sp - space_for_variables
    move $fp,$sp        # 5. $sp = $fp
    # subroutine call
    li $a0,6            # 1. No T registers to push
    jal mysub           # 2. Put argument into $a0
                          # 3. Jump and link to subroutine
    ......             # return from subroutine
    ......             # epilog
    jr $ra              # return to OS
```

Subroutine Call – Correct Answers

```assembly
.globl main
main:
    # prolog
    sub $sp,$sp,4       # 1. Push return address
    sw $ra,($sp)
    sub $sp,$sp,4       # 2. Push caller's frame pointer
    sw $fp,($sp)
    sub $sp,$fp,4       # 3. No S registers to push
    move $sp,$fp        # 4. $fp = $sp - space_for_variables
    move $fp,$sp        # 5. $sp = $fp
    # subroutine call
    li $a0,6            # 1. No T registers to push
    jal mysub           # 2. Put argument into $a0
                          # 3. Jump and link to subroutine
    ......             # return from subroutine
    ......             # epilog
    jr $ra              # return to OS
```
Prolog for mysub()

- Of course, mysub starts with a subroutine prolog. There are two variables, so space is assigned to them on the stack.

```assembly
.globl mysub
.text
.int mysub (int arg) {
    .int b,c;
    // b: 0($fp)
    // c: 4($fp)
    b = arg * 2;
    // c: 7
    c = b + 7;
    return c;
}

sub $sp,$sp,4 # 1. Push return address
sw $ra,($sp)
sub $sp,$sp,4 # 2. Push caller's frame pointer
sw $s1,$(sp)
    # 3. Push register $s1
sw $fp,$sp,8 # 4. $fp = $sp - space_for_variables
move $sp,$fp # 5. $sp = $fp
jr $ra # return to caller
```

Note that this subroutine could be written without using $s1. It is used to show how linkage works.

Using Variables

- Of course, mysub starts with a subroutine prolog. There are two variables, so space is assigned to them on the stack.

```assembly
.globl mysub
.text
.int mysub (int arg) {
    .int b,c;
    // b: 0($fp)
    // c: 4($fp)
    b = arg * 2;
    // c: 7
    c = b + 7;
    return c;
}

mul $s1,$a0,2 # arg*2
sw $s1,(__) # b = arg*2
lw $t0,(__) # b = arg*2
add $t0,($t0,(__)) # c = b + 7
sw $t0,(__) # c = b + 7
jr $ra # return to caller
```

This program is not very efficient, as written. There is no need to store and then load b. A non-optimizing compiler might do just that, however.

# body of subroutine
mul $s1,$a0,2 # arg*2
sw $s1,(__) # b = arg*2
lw $t0,(__) # b = arg*2
add $t0,($t0,(__)) # b = arg*2
sw $t0,(__) # c = b + 7
jr $ra # return to caller

Note that this subroutine could be written without using $s1. It is used to show how linkage works.
Subroutine Epilog

# body of subroutine
mul $s1,$a0,2  # arg*2
sw $s1,$0($fp)  # b = " "
lw $t0,0($fp)  # get b
add $t0,$t0,7  # b+7
sw $t0,4($fp)  # c = " "
.
.
.
jr $ra  # return to caller

# epilog
lw $v0,___(___)  # 1. Put return value in $v0
add $sp,$fp,___  # 2. $sp = $fp + space_for_variables
lw $s1,(____)  # 3. Pop register $s1
add ___ ___ ___  #
lw $fp,(____)  # 4. Pop $fp
add ___ ___ ___  #
lw $ra,(____)  # 5. Pop $ra
add ___ ___ ___  #
jr $ra  # 6. return to caller
Example Program: Factorial( N )

- The SPIM console window shows the output. The pseudo-code for the program is:

```plaintext
# main()
# {
# int a, b; // a: 0($fp), b: 4($fp)
# write("enter an int:")
# read( a );
# b = fact( a );
# write("factorial is:")
# print( b );
# }

# int fact (int n)
# {
# if ( n <= 1 )
# return 1;
# else
# return n*fact(n-1);
# }

# int fact (int n)
# {
# if ( n <= 1 )
# return 1;
# else
# return n*fact(n-1);
# }
```

- If the subroutine fact() is called with an argument greater than one, it calls itself, fact(), with a new argument. This is a new activation of the same code. This is not a problem because the data for the first activation of fact() is pushed onto the stack. The new activation has a fresh stack frame for its data.

- When the first activation gets control again, its data is available at the top of the stack. This process is illustrated at right. Each activation is represented as a green circle. Each activation works with its own data in its own stack frame. The activations do not interfere with each other.

- Each bead on the activation chain represents an activation of a subroutine. The label on a downward arc is the argument to an activation. The label on an upward arc is the returned value.

- Each bead on the activation chain corresponds to one stack frame. The picture of the stack shows what it looks like when the activation fact(1) is running.

- When the value 120 is returned to main, only main is active, the stack contains only its stack frame, and the activation chain consists only of main.
Question

- A downward arc corresponds to a _______ of one stack frame. An upward arc corresponds to a _______ of one stack frame.

A downward arc corresponds to a push of one stack frame. An upward arc corresponds to a pop of one stack frame.

Storage Class Summary

- There are three places in memory where data may be placed: in the data section (declared with .data in assembly language), on the run-time stack, and on the heap.
- A subroutine other than main() can have data in the .data section. In high-level programming languages, such as "C", this type of storage is called static.
- Variables whose storage is allocated on the run-time stack are (sometimes) called automatic variables. This is because their storage is "automatically" pushed and popped as a subroutine is entered and exited. Usually the word "variable" means "automatic variable".
- A variable whose memory is located in the heap is called a dynamic variable. These notes only briefly deal with the heap. The heap is where memory for objects is found (using the new operation in Java or C++). In "C" dynamic memory is allocated using the malloc operation.
- The heap is on top of the data segment. As dynamic variables are created it grows upward (towards the stack).
Summary

- Stack frames introduced;
- Local variables were introduced and illustrated;
- Frame pointers were introduced;
- The simple linkage convention was expanded to include the use of stack frames.