CS352 - Compilers: Principles and Practice
Homework 2 - When Stack Frames Attack
Due: 2004 November 30, beginning of lecture

1. Consider this simple C program:

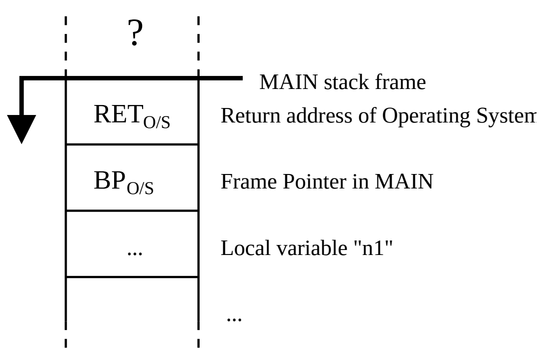
```
int fibonacci(int *n2, int *n1, int x)
{
  /* printf("fib(%d,%d,%d)\n", *n2, *n1, x); */
  int n0 = *n1 + *n2;
  *n2 = *n1;
  *n1 = n0;
  if (x <= 3) return n0;
  else return fibonacci(n2, n1, --x);
}

int main()
{
  int n1 = 1;
  int n2 = 1;
  printf("%d\n", fibonacci(&n2, &n1, 4));
}
```

The Intel x86 assembly language for this program can be found at

`http://www.cs.purdue.edu/homes/brylow/cs352/Fall2004/Homework/hw2-p1.s`

Draw the stack of activation records for this program, showing the final contents of each location when the stack is at its largest point. Label each word in the stack that you can identify. Some locations in the stack may be undefined or unlabeled.

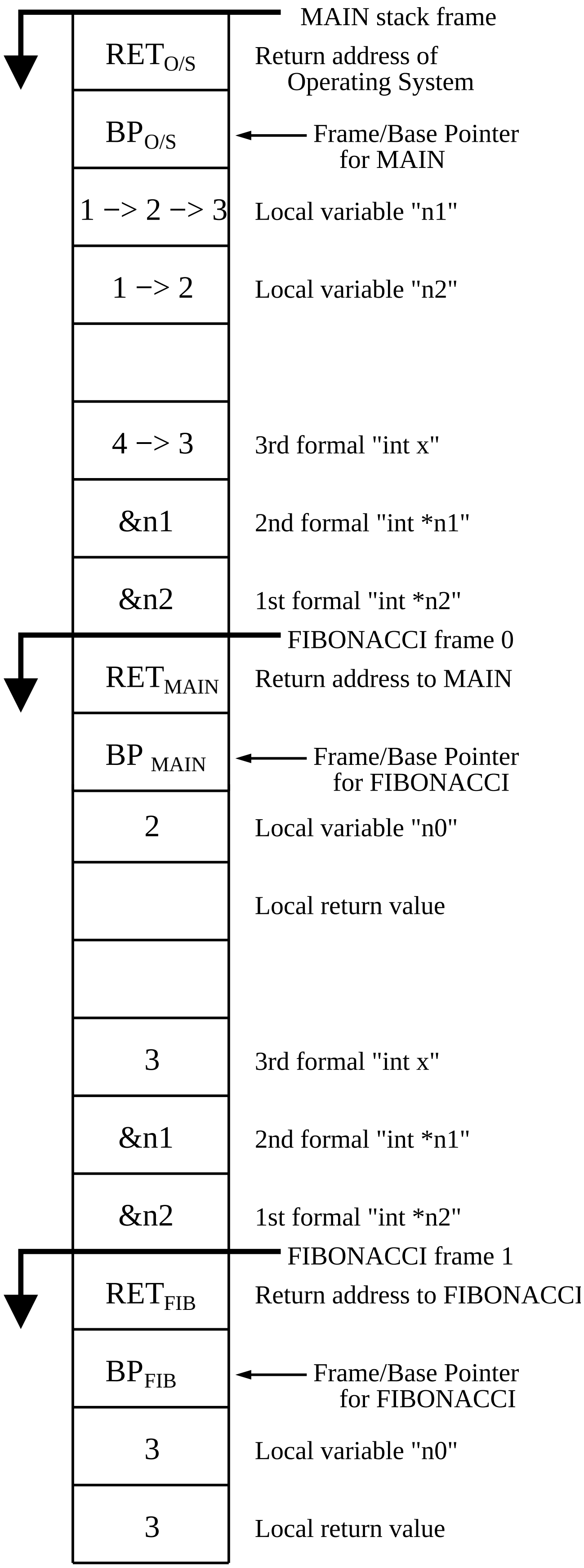
The first three words of the stack will look like this:



Fill in the rest. The Pentium architecture manuals at

`lore:/homes/cs352/Docs/Pentium-Reference-{1,2}.pdf`

may help you identify unfamiliar opcodes.


MAIN stack frame
RET O/S
Return address of Operating System
BP O/S
Frame/Base Pointer for MAIN
1 -> 2 -> 3
Local variable "n1"
1 -> 2
Local variable "n2"
4 -> 3
3rd formal "int x"
&n1
2nd formal "int *n1"
&n2
1st formal "int *n2"
FIBONACCI frame 0
RET MAIN
Return address to MAIN
BP MAIN
Frame/Base Pointer for FIBONACCI
2
Local variable "n0"
Local return value
3
3rd formal "int x"
&n1
2nd formal "int *n1"
&n2
1st formal "int *n2"
FIBONACCI frame 1
RET FIB
Return address to FIBONACCI
BP FIB
Frame/Base Pointer for FIBONACCI
3
Local variable "n0"
3
Local return value

2. The source code from problem 1 can be downloaded from

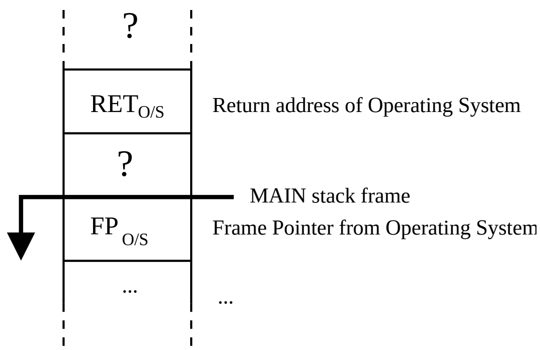
   `http://www.cs.purdue.edu/homes/brylow/cs352/Fall2004/Homework/hw2-p1.c`

   Login to Seven.cs.purdue.edu and run gcc to generate the PowerPC assembly language for hw2-p1.c.

   ```
   gcc -S hw2-p1.c
   ```

   Draw the stack of activation records for this program, showing the final contents of each location when the stack is at its largest point. Label each word in the stack that you can identify. Some locations in the stack may be undefined or unlabeled.

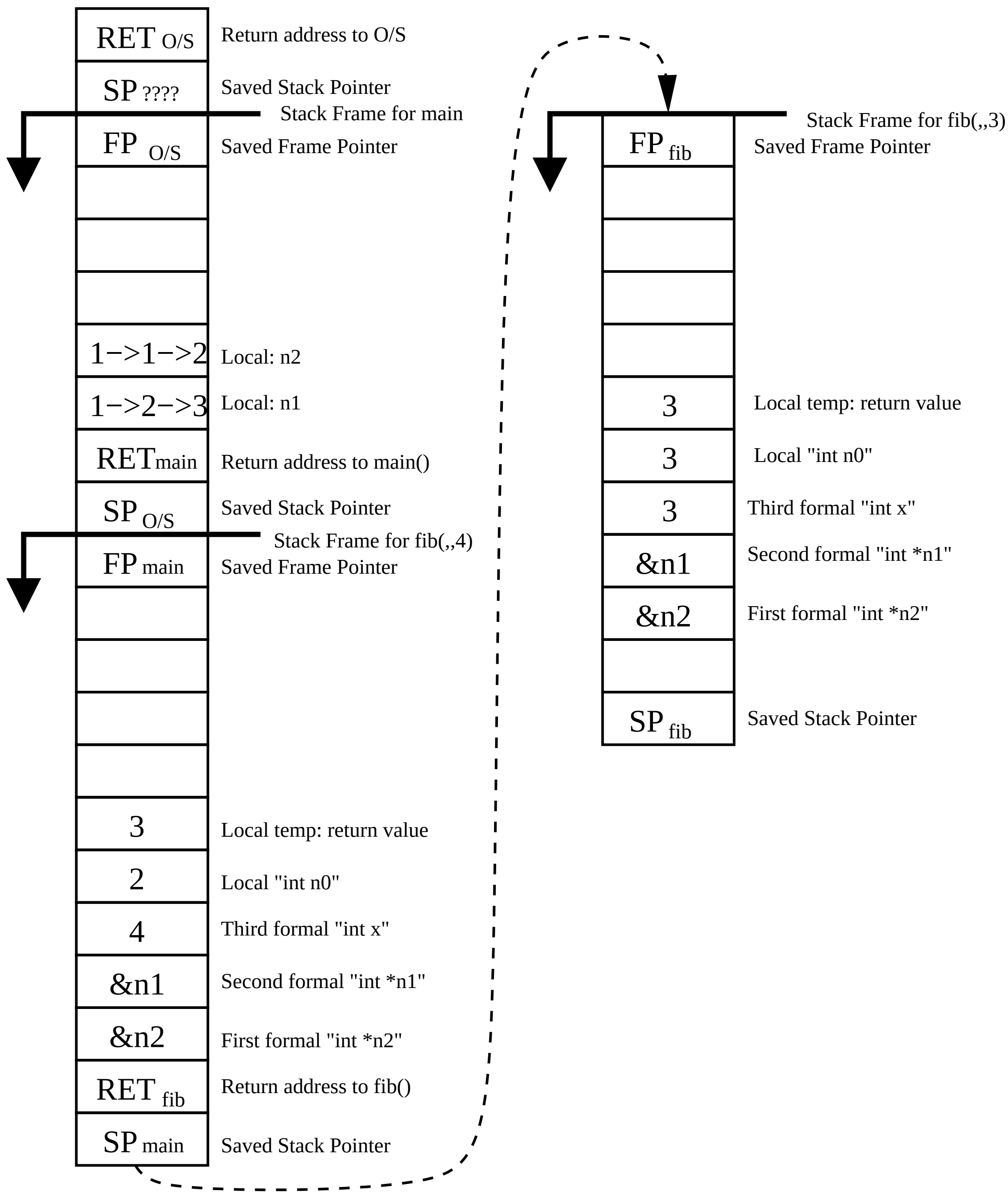
   The first three words of the stack will look like this:



   Fill in the rest. The PowerPC architecture manual at

   `lore:/homes/cs352/Docs/PowerPC-Reference.pdf`

   may help you identify unfamiliar opcodes.


RET O/S
Return address to O/S
SP ????
Saved Stack Pointer
Stack Frame for main
FP O/S
Saved Frame Pointer
1−>1−>2
Local: n2
1−>2−>3
Local: n1
RETmain
Return address to main()
SP O/S
Saved Stack Pointer
Stack Frame for fib(,,4)
FP main
Saved Frame Pointer
3
Local temp: return value
2
Local "int n0"
4
Third formal "int x"
&n1
Second formal "int *n1"
&n2
First formal "int *n2"
RET fib
Return address to fib()
SP main
Saved Stack Pointer
Stack Frame for fib(,,3)
FP fib
Saved Frame Pointer
3
Local temp: return value
3
Local "int n0"
3
Third formal "int x"
&n1
Second formal "int *n1"
&n2
First formal "int *n2"
SP fib
Saved Stack Pointer

3. (a) By convention, the PowerPC machine prefers to pass function arguments in designated argument registers. Why does the program in question 2 store the values of `fibonacci`'s arguments on the stack during execution?

   Because `fibonacci` potentially calls itself (and thus is not a *leaf* method) the formal parameters to the function must be saved to the stack frame in order to preserve their values across any subsequent calls.

   (b) Give an example of a C program that requires the PowerPC machine to *spill* function arguments into the stack frame. Verify your example with the PowerPC compiler. Where does gcc put these arguments on the stack in relation to your callee's stack frame?

   Any program containing a function with more than eight arguments will demonstrate argument spilling.

   Gcc puts these arguments above the callee's stack frame, near the bottom of the caller's stack frame.

   (c) Each PowerPC routine begins with a line like, "`stwu 1, -x(1)`", where x is the size of the stack frame to allocate. What formula does gcc seem to be using to calculate x?

   The base stack frame size is 24 bytes – room to save 6 words, like the frame pointer, stack pointer, return address, and others. The frame includes a word for each local variable. The frame includes a word for each parameter passed to a callee past the first 8 formals that can be fit into registers. PowerPC prefers to load memory in 16-byte multiples, so gcc always rounds up to the next highest multiple.

   The formula:
   (24 bytes
   +(locals $\times$ 4 bytes)
   +(callee parameters past 8 $\times$ 4 bytes))
   rounded up to multiple of 16.