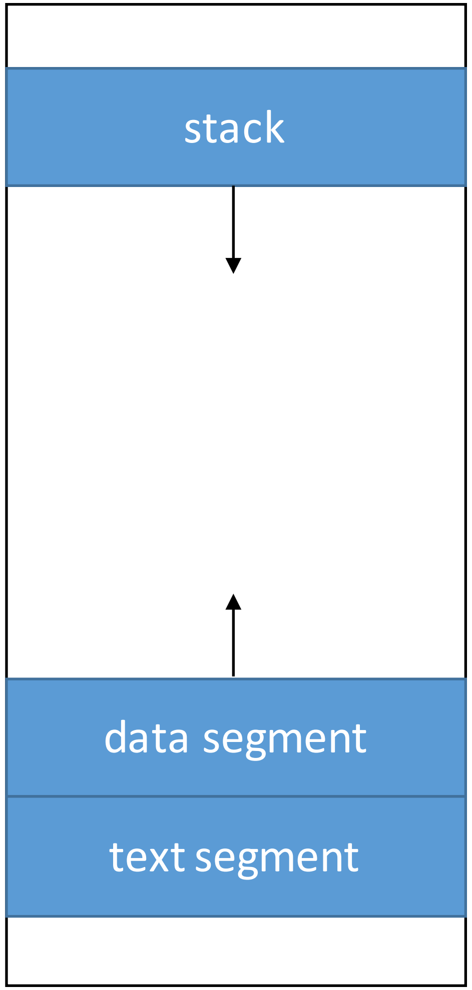
Review: Stack Frame

CS451 ’16 Spring

Process memory layout

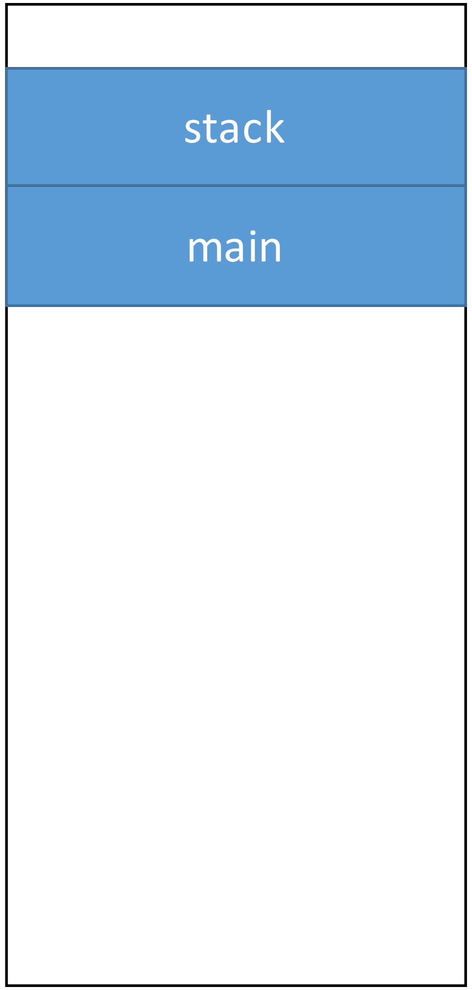


Stack is used to store data associated with function calls

Data stores static data where values can change. On top of it, dynamic data is allocated with *malloc()*

Text is for machine language of the user program

Call stack

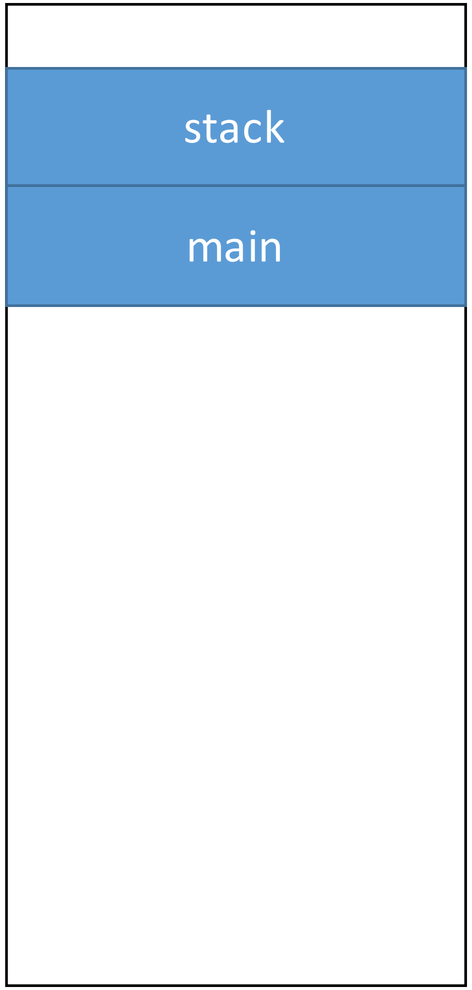



```
int main (int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f (int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g (int param) {
    return param * 2;
}
```

Call stack

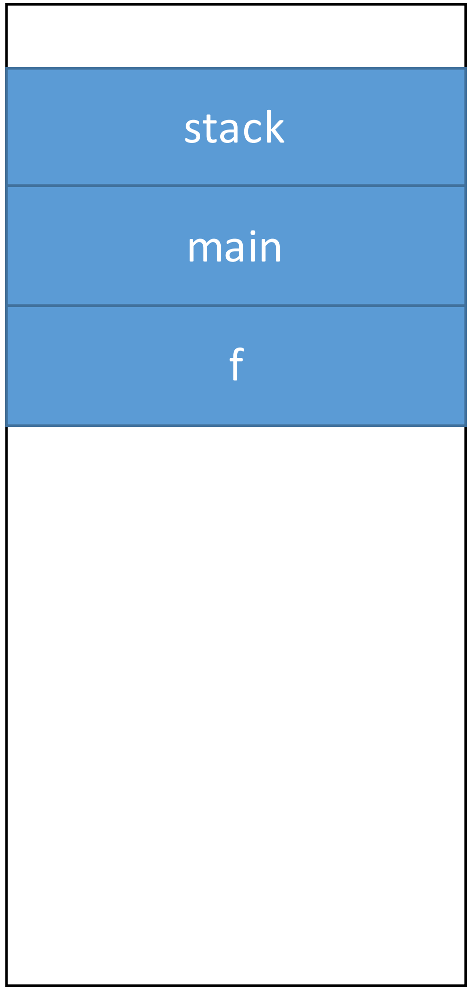



```
int main (int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f (int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g (int param) {
    return param * 2;
}
```

Call stack

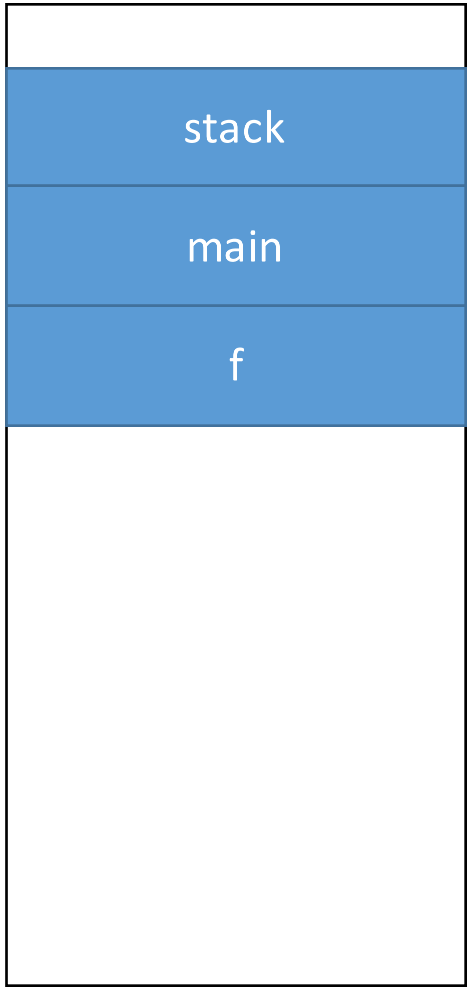



```
int main (int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f (int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g (int param) {
    return param * 2;
}
```

Call stack




```
int main (int argc, char **argv) {
   int n1 = f(3, -5);
   n1 = g(n1);
}

int f (int p1, int p2) {
   int x;
   int a[3];
   ...
   x = g(a[2]);
   return x;
}

int g (int param) {
   return param * 2;
}
```

Call stack


```
int main (int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f (int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g (int param) {
    return param * 2;
}
```

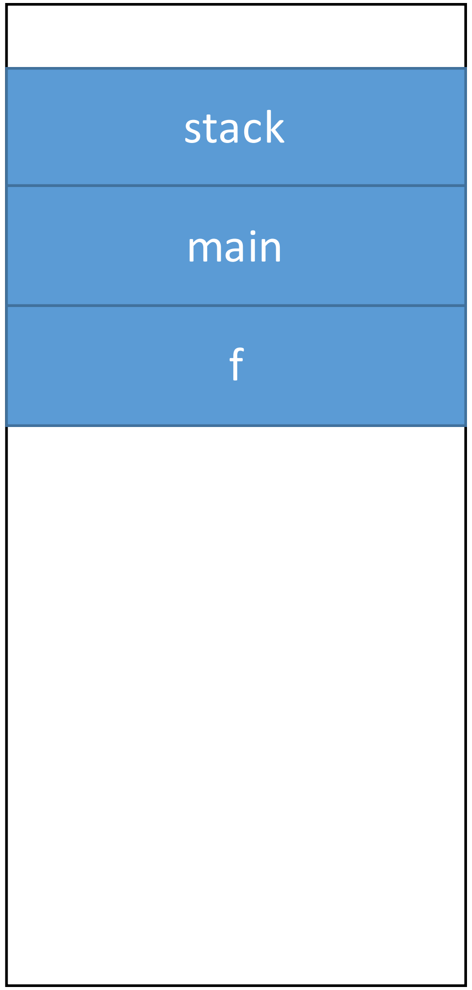
Call stack


```
int main (int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f (int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g (int param) {
    return param * 2;
}
```

Call stack

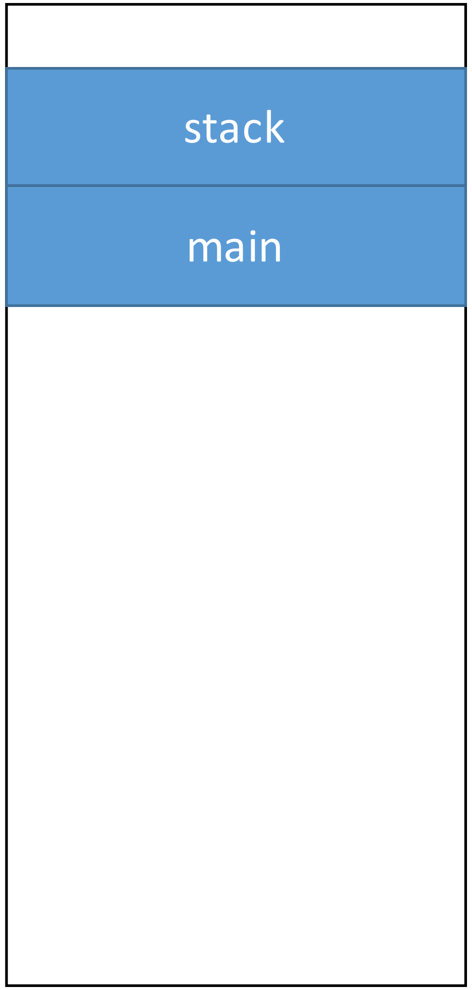



```
int main (int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f (int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g (int param) {
    return param * 2;
}
```

Call stack

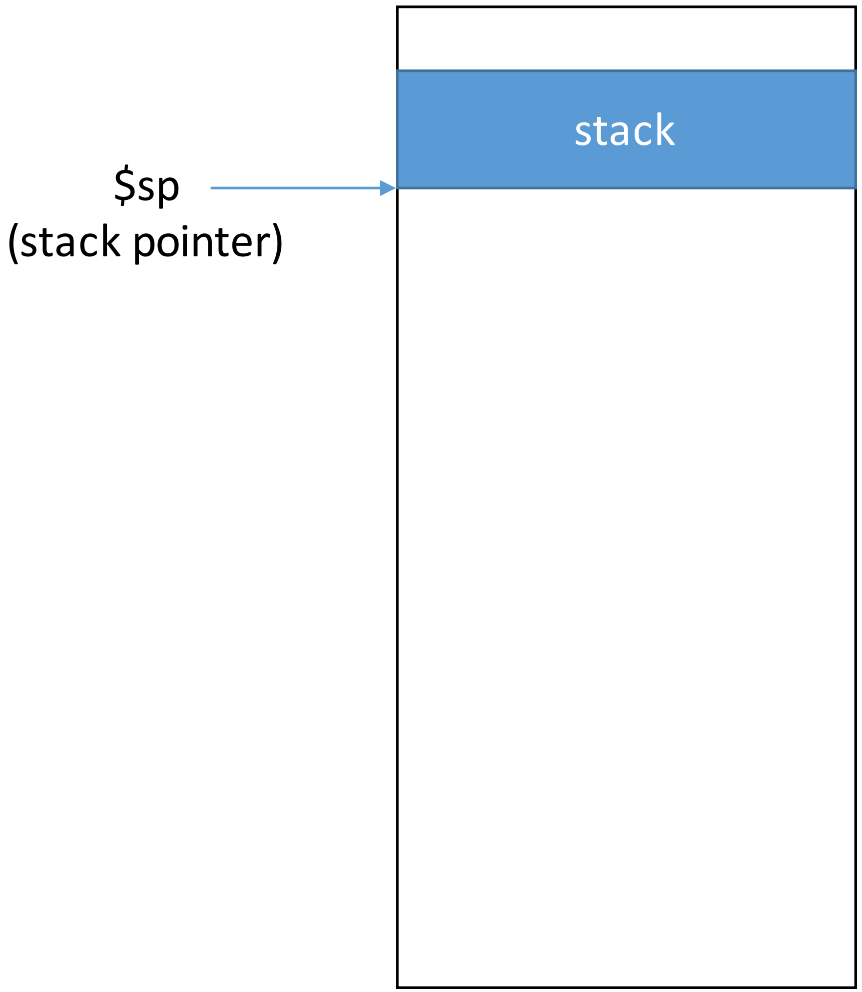



```
int main (int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f (int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g (int param) {
    return param * 2;
}
```

Call stack – what should we store in a frame?

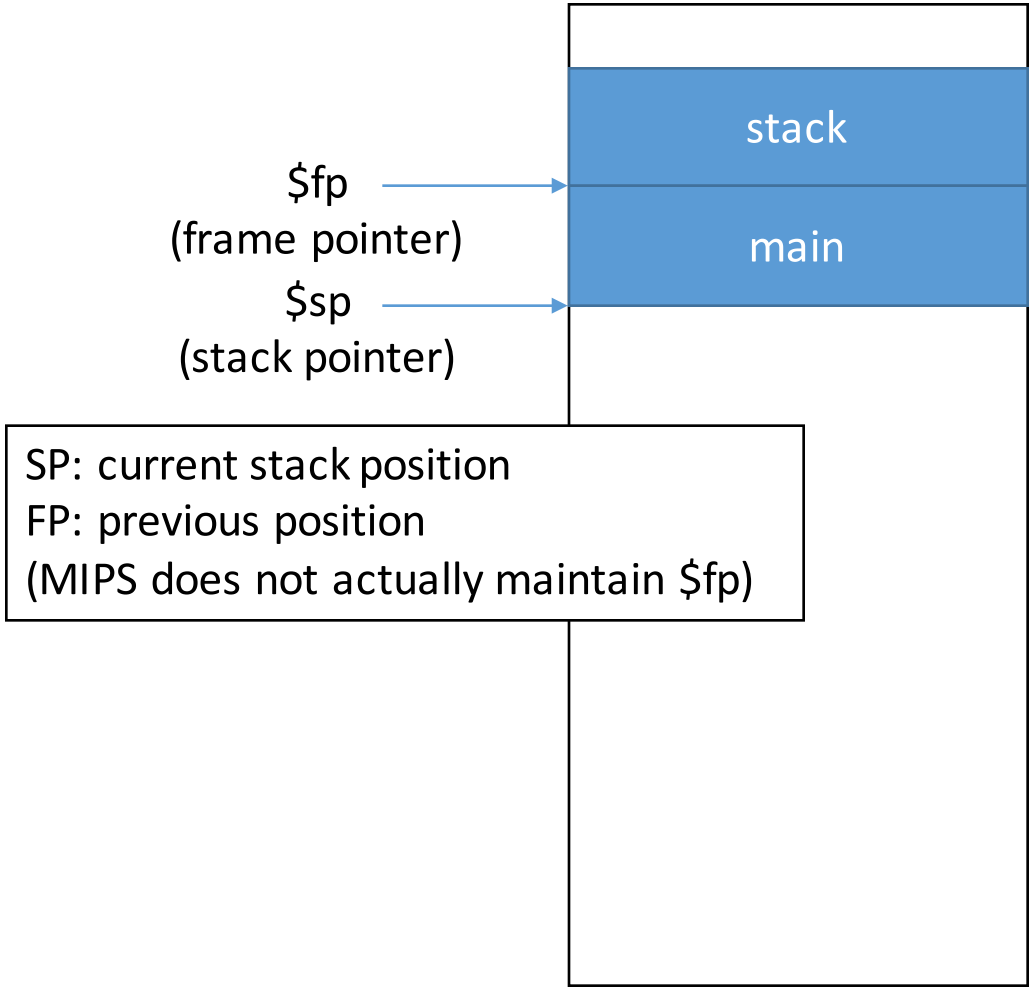



```
int main (int argc, char **argv) {
   int a, b;
   n1 = f(3, -5, 1, 4, 6);
   n1 = g(n1);
}

int f (int p1, int p2, int p3, int p4, int p5) {
   int x = 3;
   x = g(p1);
   return x;
}

int g (int param) {
   return param * 2;
}
```

Call stack – what should we store in a frame?

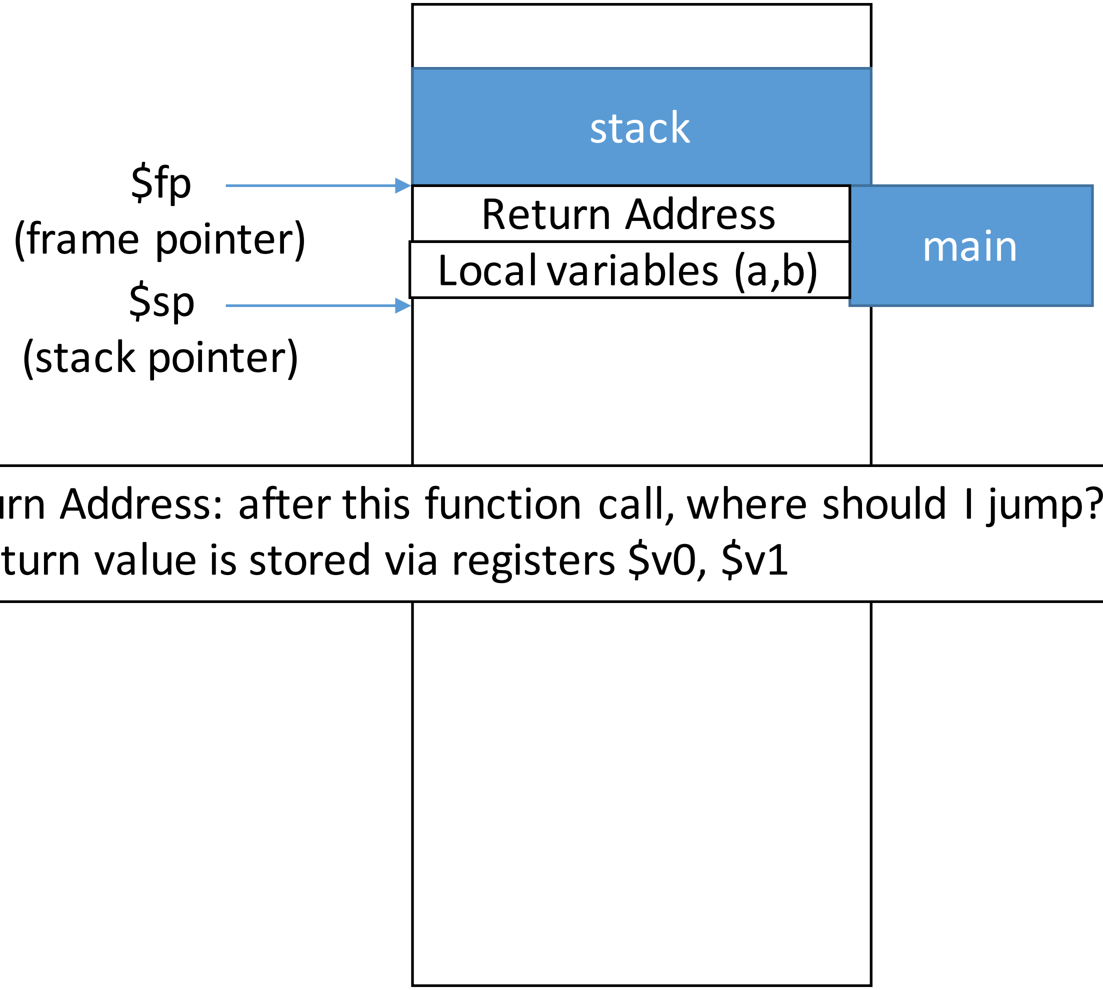



```
int main (int argc, char **argv) {
    int a, b;
    n1 = f(3, -5, 1, 4, 6);
    n1 = g(n1);
}

int f (int p1, int p2, int p3, int p4, int p5) {
    int x = 3;
    x = g(p1);
    return x;
}

int g (int param) {
    return param * 2;
}
```

Call stack – what should we store in a frame?



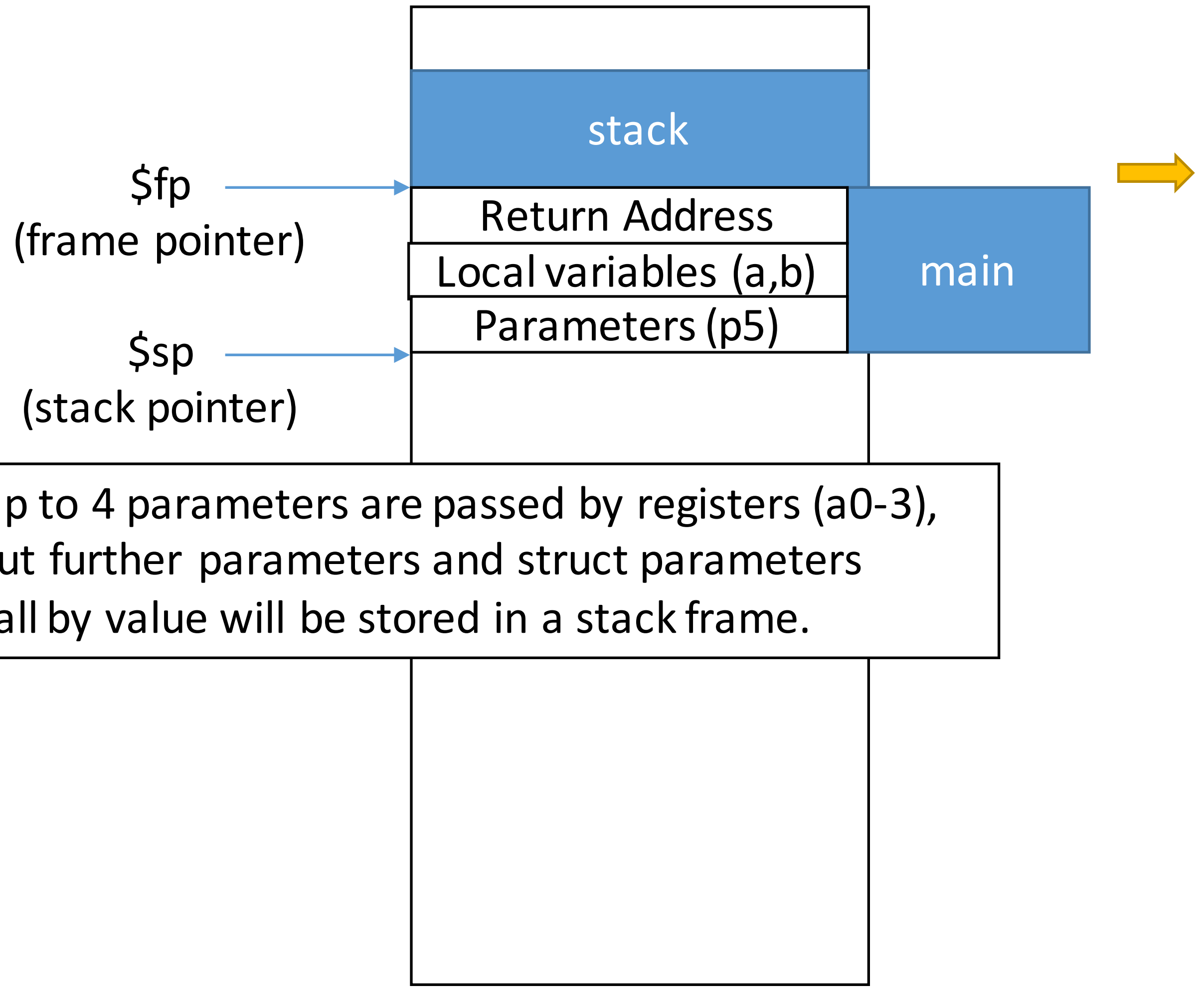
Return Address: after this function call, where should I jump?
* Return value is stored via registers $v0, $v1

```
int main (int argc, char **argv) {
    int a, b;
    n1 = f(3, -5, 1, 4, 6);
    n1 = g(n1);
}

int f (int p1, int p2, int p3, int p4, int p5) {
    int x = 3;
    x = g(p1);
    return x;
}

int g (int param) {
    return param * 2;
}
```

Call stack – what should we store in a frame?



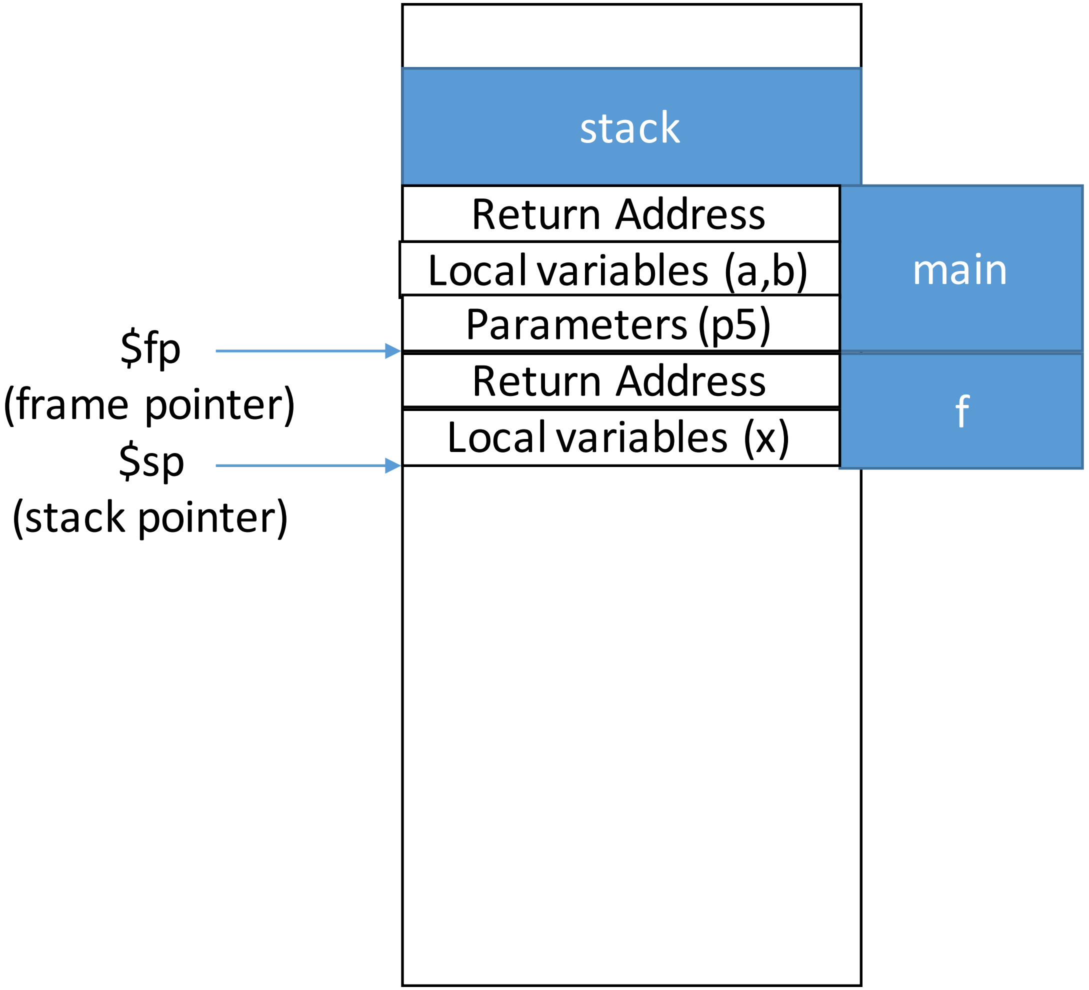
Up to 4 parameters are passed by registers (a0-3), but further parameters and struct parameters call by value will be stored in a stack frame.

```
int main (int argc, char **argv) {
    int a, b;
    n1 = f(3, -5, 1, 4, 6);
    n1 = g(n1);
}

int f (int p1, int p2, int p3, int p4, int p5) {
    int x = 3
    x = g(p1);
    return x;
}

int g (int param) {
    return param * 2;
}
```

Call stack – what should we store in a frame?

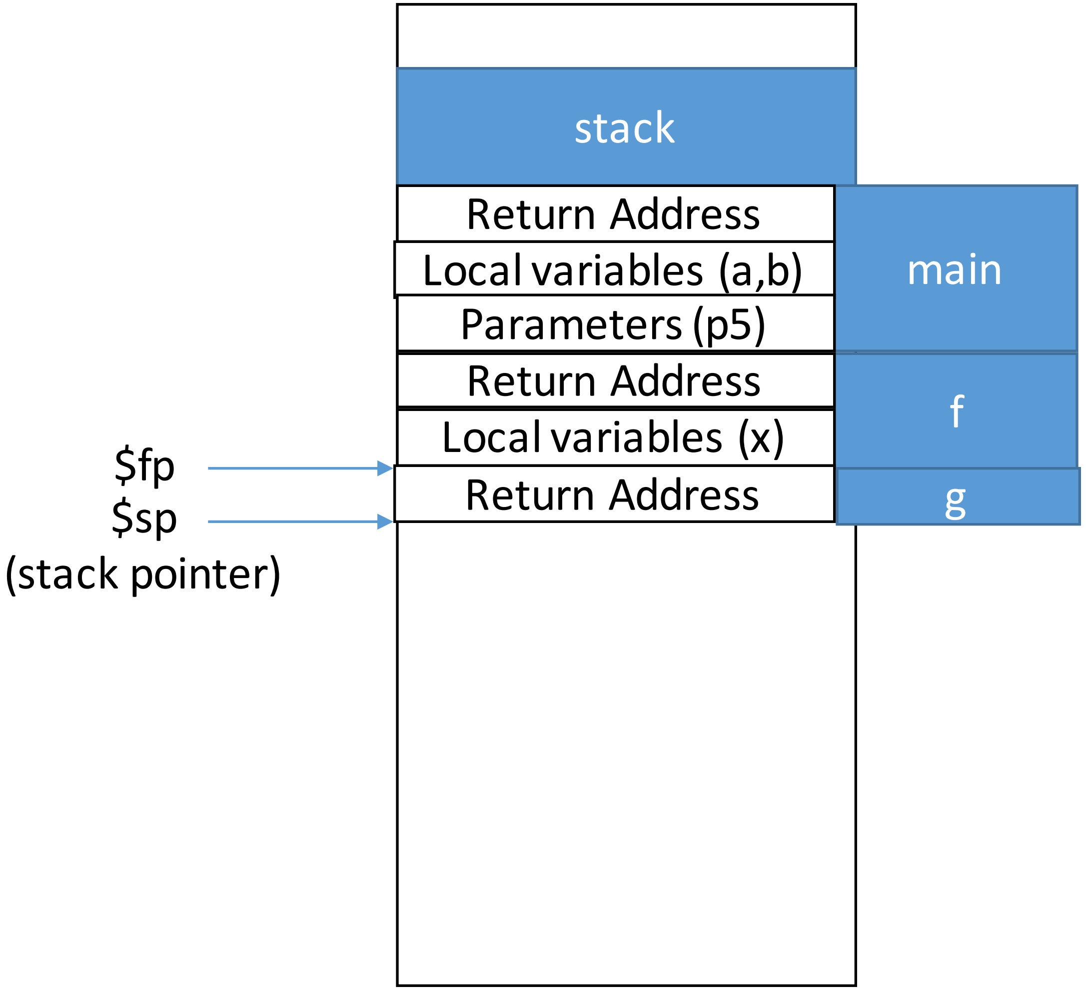



```
int main (int argc, char **argv) {
    int a, b;
    n1 = f(3, -5, 1, 4, 6);
    n1 = g(n1);
}

int f (int p1, int p2, int p3, int p4, int p5) {
    int x = 3
    x = g(p1);
    return x;
}

int g (int param) {
    return param * 2;
}
```

Call stack – what should we store in a frame?




```
int main (int argc, char **argv) {
    int a, b;
    n1 = f(3, -5, 1, 4, 6);
    n1 = g(n1);
}

int f (int p1, int p2, int p3, int p4, int p5) {
    int x = 3
    x = g(p1);
    return x;
}

int g (int param) {
    return param * 2;
}
```

MIPS https://courses.cs.washington.edu/courses/cse451/16sp/help/mips.html

- RISC instruction set architecture (e.g., ARM)
- System/161 uses a dialect of MIPS processor: MIPS-161
- Registers are 32-bit
- Some example instructions
  - add $t0, $t1, $t2 # $t0 = $t1 + $t2 (add as signed integers)
  - move $t2, $t3 # $t2 = $t3
  - lw register, RAM_source # copy word at source to register
  - sw register, RAM_destination # copy word at register to destination
  - …
  - See http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm

MIPS example

```
int f (int a, int b) {
    return a+b;
}

int g (int a, int b) {
    return f(b, a)
}
```

```
f:  add   $v0, $a0, $a1
    jr    $ra              # jump to return address

g:  addi  $sp, $sp, -12    # allocate stack space for 3 words
    sw    $ra, 8($sp)      # store return address
    sw    $a0, 4($sp)
    sw    $a1, 0($sp)      # store local variable (prev. param)
    lw    $a0, 0($sp)      # 1st arg. = b
    lw    $a1, 4($sp)      # 2nd arg. = a
    jal   f                # jump to f, stores the next inst. to $ra
    lw    $a0, 4($sp)      # restore a
    lw    $a1, 0($sp)      # restore b
    lw    $ra, 8($sp)      # restore return address
    addi  $sp, $sp, 12     # pop stack frame
    jr    $ra              # jump to return address
```

MIPS more example

- Delayed load, delayed branch
  - Because of CPU pipeline, load and branch instructions are in effect in the one instruction later.
  - i.e., the next instruction cannot use loaded register immediately
  - and, the next instruction of branch always is executed regardless of branching

```
lw    v0, 4(v1)
jr    v0              # wants to jump v0, but it's not actually loaded
```

The simplest solution is to put **nop** after load/branch

```
lw    v0, 4(v1)
nop
jr    v0
```

MIPS more & more example

- MIPS coprocessor
  - Coprocessor 0 provides an abstraction of the function necessary to support an OS: exception handling, memory management, scheduling, and control of critical resources[1].
  - Registers include ones related to TLB, page table, and etc.
    - Accessed by mfc0 (move from coprocessor 0), mtc0 (move to …)
    - e.g., mfc $k0, c0_cause # load the exception cause to k0
- k0, k1 registers
  - designated general purpose registers (not in coprocessor) for kernel-use
  - user code should not rely on these
- You will see these in the later project!

[1]http://www.cs.cornell.edu/courses/cs3410/2015sp/MIPS_Vol3.pdf