Assembler 2

- Procedures
- Stack Frames
- Spilling

Procedures

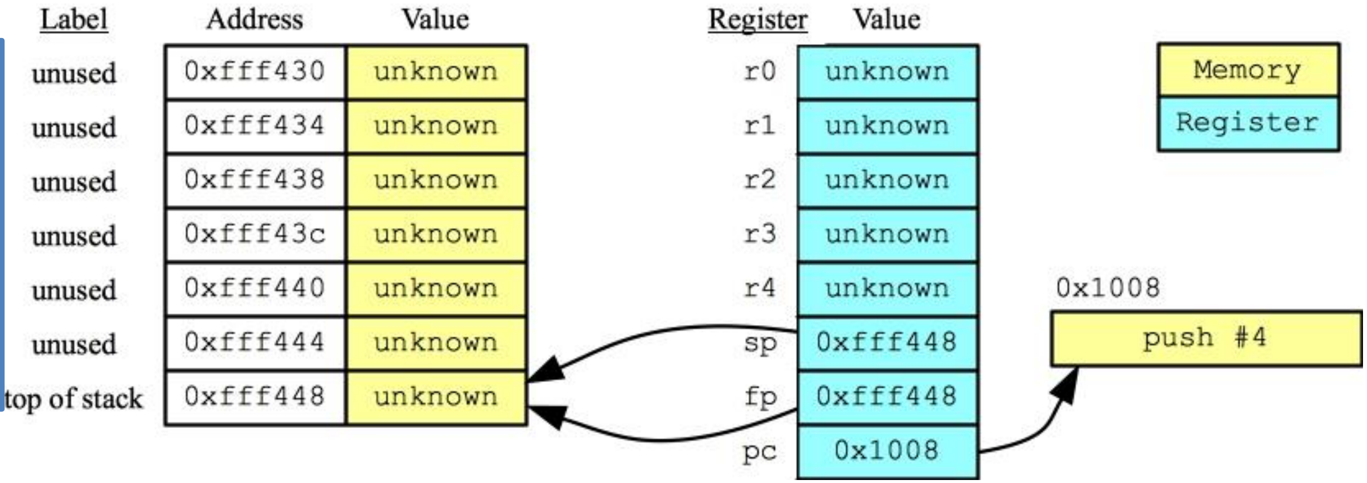
C code	Assembly code
int a()	a:
{	mov #1 -> %r0
return 1;	ret
}	
main()	main:
{	push #4
int i;	jsr a
	st %r0 -> [fp]
i = a();	ret
}	

- **Main()** first allocates one variable on the stack, and then calls "**jsr a**", which means jump to subroutine **a**.
- All **a()** does is return 1 to its caller -- it does that by setting **r0** to one, and then calling **"ret"**.
- When control returns to **main()** it stores **a's** return value, which is in **r0**, to the memory that it has allocated for **i**. And it returns.

Procedures

Before **program starts**

```
a:
            mov #1 -> %r0
            ret

main:
            push #4
            jsr a
            st %r0 -> [fp]
            ret
```

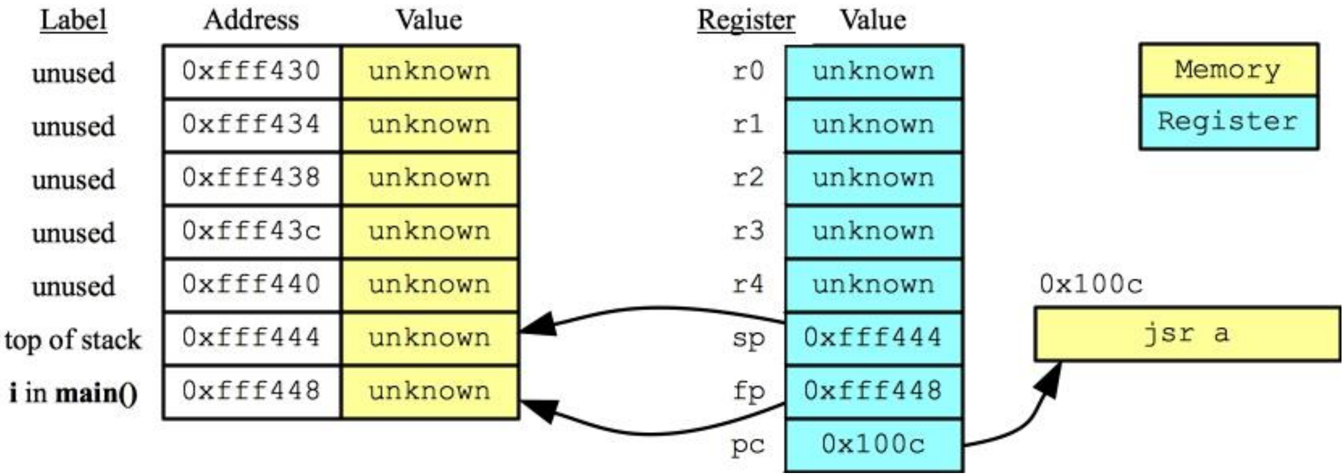


Label	Address	Value
unused	0xfff430	unknown
unused	0xfff434	unknown
unused	0xfff438	unknown
unused	0xfff43c	unknown
unused	0xfff440	unknown
unused	0xfff444	unknown
top of stack	0xfff448	unknown

Register	Value
r0	unknown
r1	unknown
r2	unknown
r3	unknown
r4	unknown
sp	0xfff448
fp	0xfff448
pc	0x1008

After **push #4** executed

```
a:
            mov #1 -> %r0
            ret

main:
            push #4
            jsr a
            st %r0 -> [fp]
            ret
```



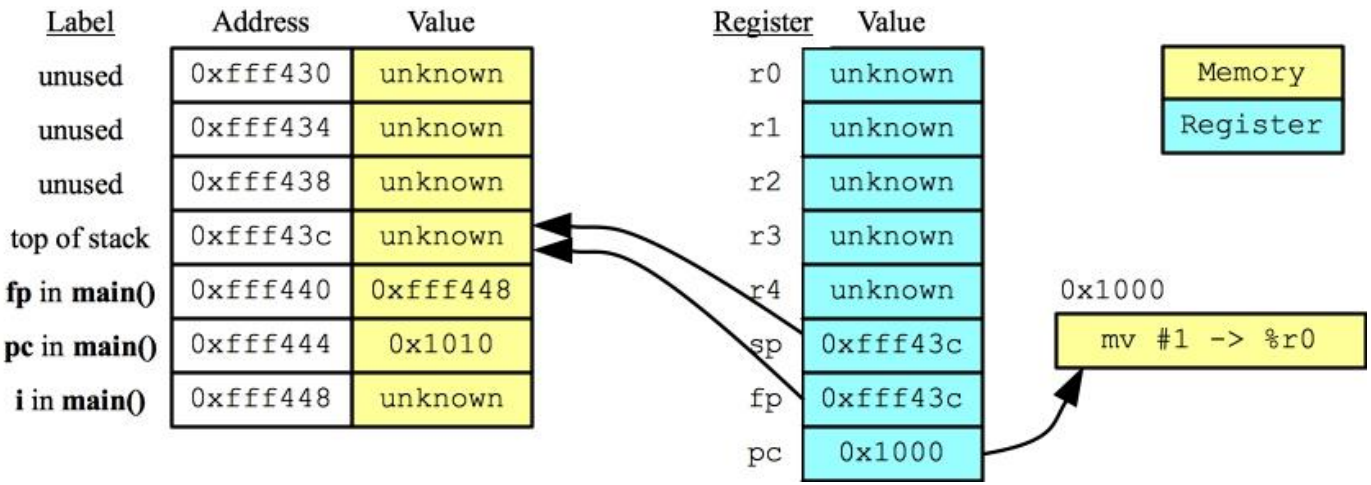
Label	Address	Value
unused	0xfff430	unknown
unused	0xfff434	unknown
unused	0xfff438	unknown
unused	0xfff43c	unknown
unused	0xfff440	unknown
top of stack	0xfff444	unknown
i in **main()**	0xfff448	unknown

Register	Value
r0	unknown
r1	unknown
r2	unknown
r3	unknown
r4	unknown
sp	0xfff444
fp	0xfff448
pc	0x100c

Procedures

After jsr a executed

```
a:
        mov #1 -> %r0
        ret

main:
        push #4
        jsr a
        st %r0 -> [fp]
        ret
```

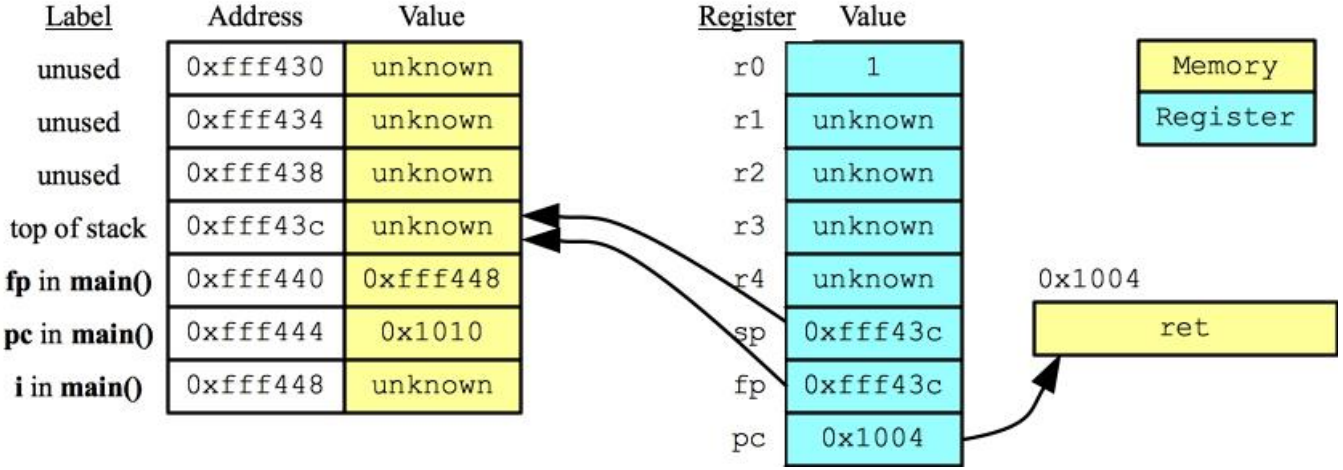


Label	Address	Value
unused	0xfff430	unknown
unused	0xfff434	unknown
unused	0xfff438	unknown
top of stack	0xfff43c	unknown
fp in **main()**	0xfff440	0xfff448
pc in **main()**	0xfff444	0x1010
i in **main()**	0xfff448	unknown

Register	Value
r0	unknown
r1	unknown
r2	unknown
r3	unknown
r4	unknown
sp	0xfff43c
fp	0xfff43c
pc	0x1000

After mov #1 -> %r0 executed

```
a:
        mov #1 -> %r0
        ret

main:
        push #4
        jsr a
        st %r0 -> [fp]
        ret
```



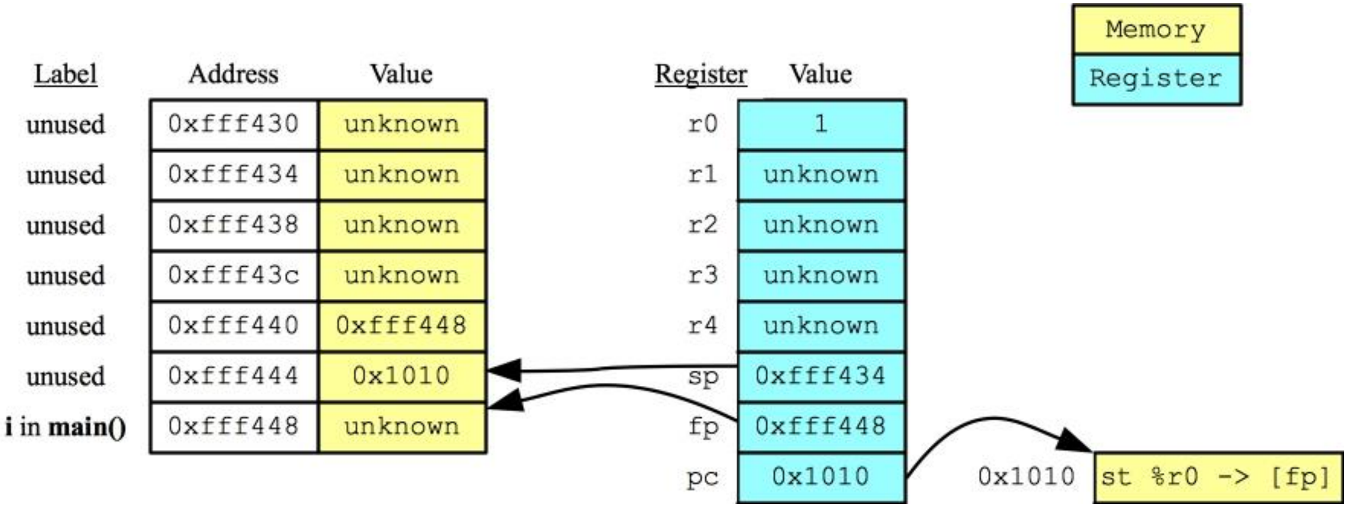
Label	Address	Value
unused	0xfff430	unknown
unused	0xfff434	unknown
unused	0xfff438	unknown
top of stack	0xfff43c	unknown
fp in **main()**	0xfff440	0xfff448
pc in **main()**	0xfff444	0x1010
i in **main()**	0xfff448	unknown

Register	Value
r0	1
r1	unknown
r2	unknown
r3	unknown
r4	unknown
sp	0xfff43c
fp	0xfff43c
pc	0x1004

Procedures

After **ret** executed

```
a:
        mov #1 -> %r0
        ret

main:
        push #4
        jsr a
        st %r0 -> [fp]
        ret
```

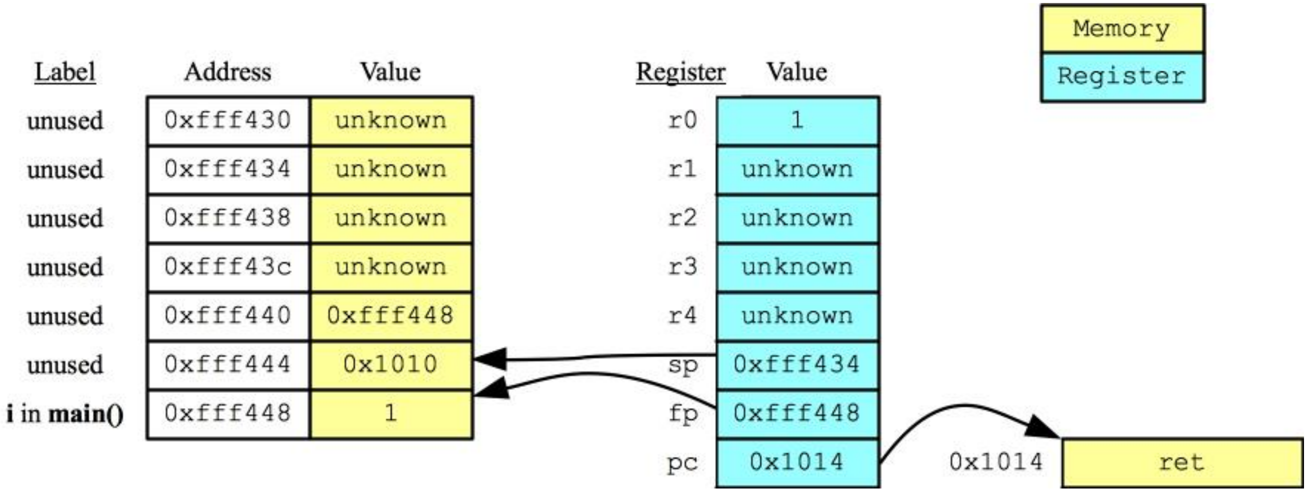


Label	Address	Value
unused	0xfff430	unknown
unused	0xfff434	unknown
unused	0xfff438	unknown
unused	0xfff43c	unknown
unused	0xfff440	0xfff448
unused	0xfff444	0x1010
i in main()	0xfff448	unknown

Register	Value
r0	1
r1	unknown
r2	unknown
r3	unknown
r4	unknown
sp	0xfff434
fp	0xfff448
pc	0x1010

After **st %r0->[fp]** executed

```
a:
        mov #1 -> %r0
        ret

main:
        push #4
        jsr a
        st %r0 -> [fp]
        ret
```



Label	Address	Value
unused	0xfff430	unknown
unused	0xfff434	unknown
unused	0xfff438	unknown
unused	0xfff43c	unknown
unused	0xfff440	0xfff448
unused	0xfff444	0x1010
i in main()	0xfff448	1

Register	Value
r0	1
r1	unknown
r2	unknown
r3	unknown
r4	unknown
sp	0xfff434
fp	0xfff448
pc	0x1014

Procedures with local parameters

C code

```
int a(int i)
{
  int j;

  j = i+1;
  return j;
}

main()
{
  int i;

  i = a(5);
}
```

Assembly code

```
a:
     push #4
     ld [fp+12] -> %r0   /i
     add %r0, %g1 -> %r0 /r0=i+1
     st %r0 -> [fp]      /j=r0
     ld [fp] -> %r0    /return
     ret
main:
     push #4
     mov #5 -> %r0
     st %r0 -> [sp]--
     jsr a
     pop #4
     st %r0 -> [fp]
     ret
```

Prodedures with local parameters

C code

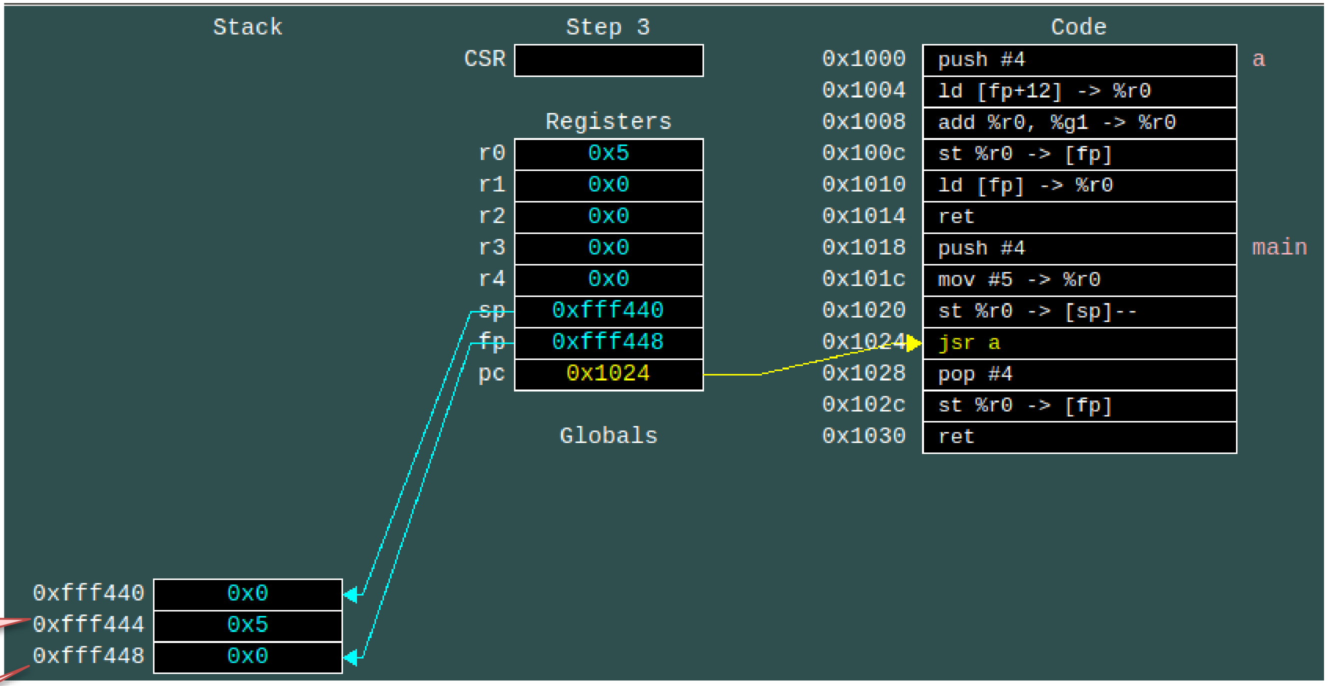
```c
int a(int i)
{
  int j;

  j = i+1;
  return j;
}

main()
{
  int i;

  i = a(5);
}
```

Assembly code

```
a:
        push #4
        ld [fp+12] -> %r0    /i
        add %r0, %g1 -> %r0 /r0=i+1
        st %r0 -> [fp]       /j=r0
        ld [fp] -> %r0     /return
        ret
main:
        push #4
        mov #5 -> %r0
        st %r0 -> [sp]--
        jsr a
        pop #4
        st %r0 -> [fp]
        ret
```

Procedures with local parameters

Before executing *jsr a* line



```
          Stack                          Step 3                          Code
                                   CSR                        0x1000  push #4                   a
                                                              0x1004  ld [fp+12] -> %r0
                                       Registers              0x1008  add %r0, %g1 -> %r0
                                   r0  0x5                    0x100c  st %r0 -> [fp]
                                   r1  0x0                    0x1010  ld [fp] -> %r0
                                   r2  0x0                    0x1014  ret
                                   r3  0x0                    0x1018  push #4                   main
                                   r4  0x0                    0x101c  mov #5 -> %r0
                                   sp  0xfff440               0x1020  st %r0 -> [sp]--
                                   fp  0xfff448               0x1024  jsr a
                                   pc  0x1024                 0x1028  pop #4
                                                              0x102c  st %r0 -> [fp]
                                        Globals               0x1030  ret

0xfff440  0x0
0xfff444  0x5
0xfff448  0x0
```

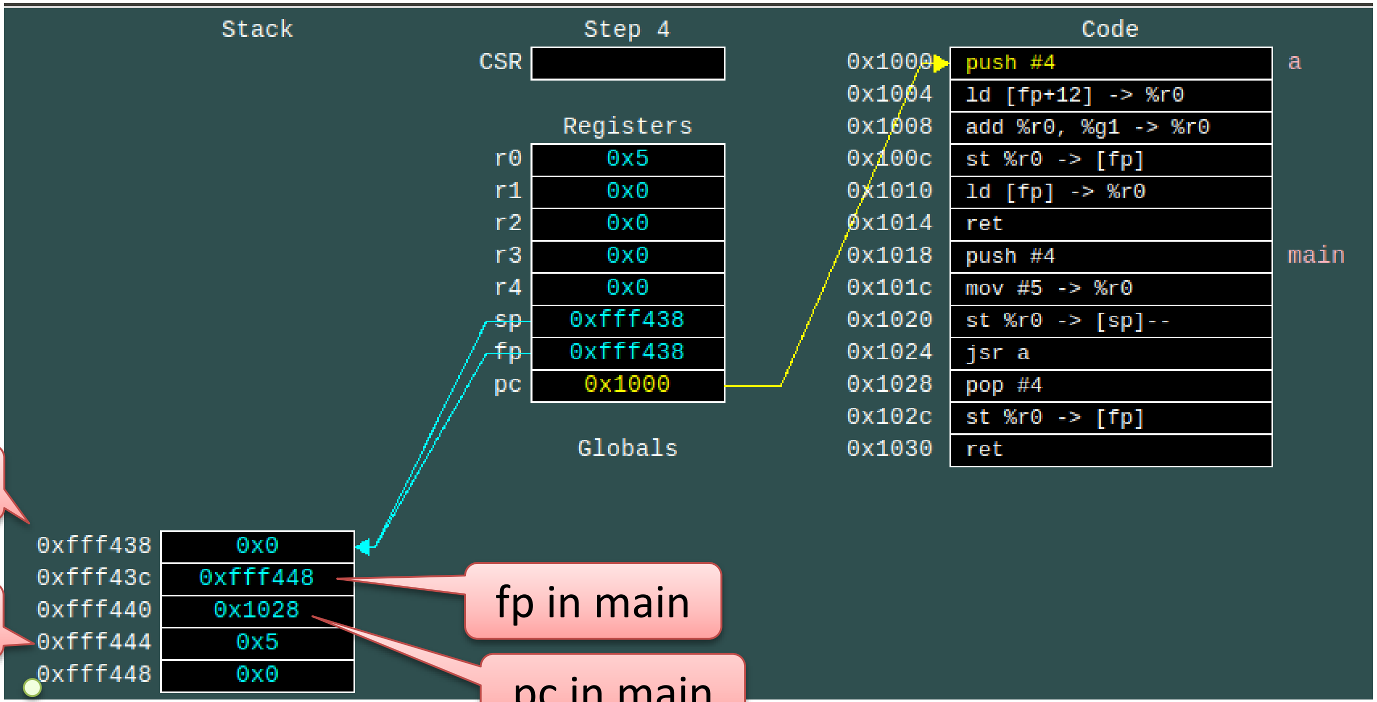
fp-4

fp

```
int a(int i)
{
   int j;

   j = i+1;
   return j;
}

main()
{
   int i;

   i = a(5);
}
```

Procedures with local parameters

After executing *jsr a* line



```
                 Step 4
            CSR [        ]

               Registers
            r0   0x5
            r1   0x0
            r2   0x0
            r3   0x0
            r4   0x0
            sp   0xfff438
            fp   0xfff438
            pc   0x1000

                Globals
```

```
                 Code
0x1000   push #4                a
0x1004   ld [fp+12] -> %r0
0x1008   add %r0, %g1 -> %r0
0x100c   st %r0 -> [fp]
0x1010   ld [fp] -> %r0
0x1014   ret
0x1018   push #4                main
0x101c   mov #5 -> %r0
0x1020   st %r0 -> [sp]--
0x1024   jsr a
0x1028   pop #4
0x102c   st %r0 -> [fp]
0x1030   ret
```

```
          Stack
0xfff438   0x0
0xfff43c   0xfff448
0xfff440   0x1028
0xfff444   0x5
0xfff448   0x0
```

fp

fp+12

fp in main

pc in main

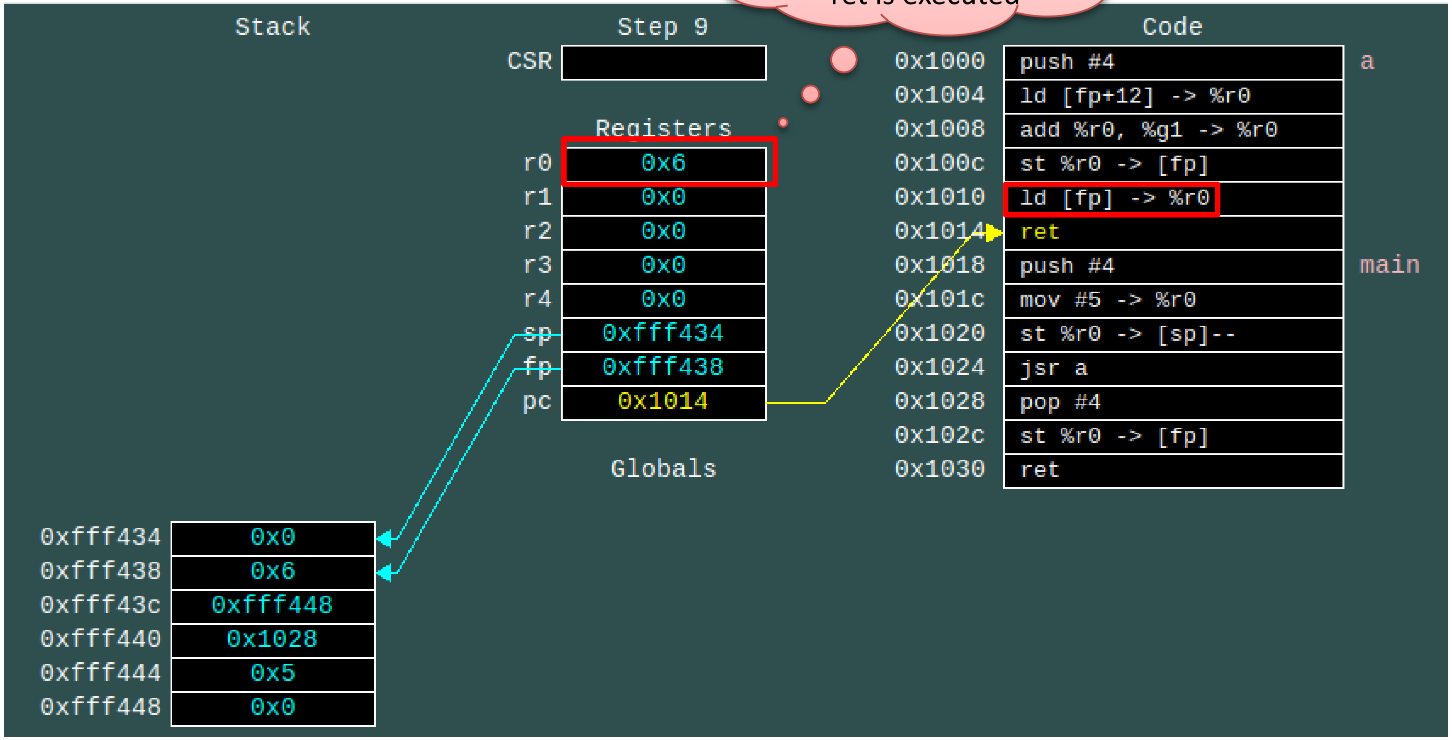
Address of the local parameter in a(**int i**)

```
int a(int i)
{
   int j;

   j = i+1;
   return j;
}

main()
{
   int i;

   i = a(5);
}
```

Procedures with local parameters

Before executing *ret* in a

Return value is written to r0 before ret is executed



Stack | Step 9 | Code

CSR

Registers

r0	0x6
r1	0x0
r2	0x0
r3	0x0
r4	0x0
sp	0xfff434
fp	0xfff438
pc	0x1014

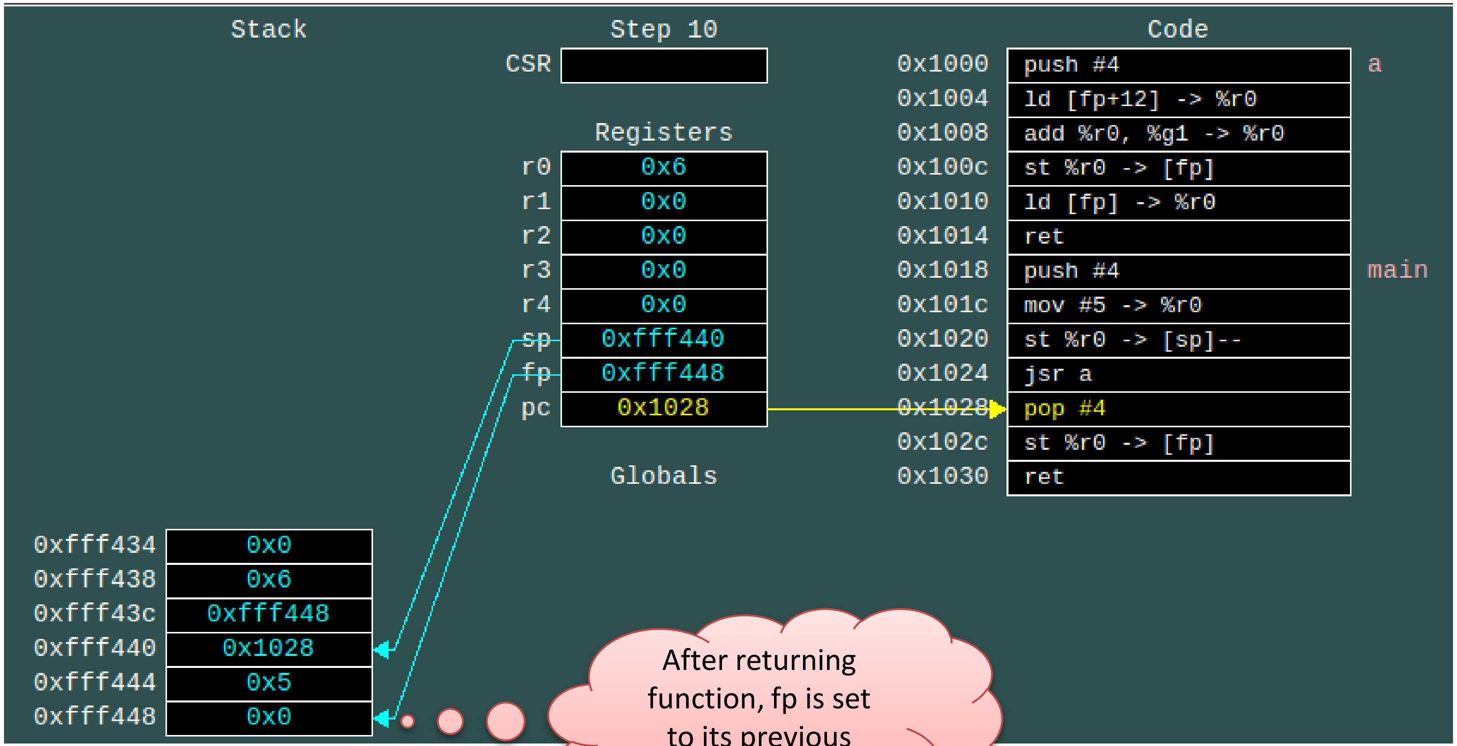
Globals

0x1000	push #4	a
0x1004	ld [fp+12] -> %r0	
0x1008	add %r0, %g1 -> %r0	
0x100c	st %r0 -> [fp]	
0x1010	ld [fp] -> %r0	
0x1014	ret	
0x1018	push #4	main
0x101c	mov #5 -> %r0	
0x1020	st %r0 -> [sp]--	
0x1024	jsr a	
0x1028	pop #4	
0x102c	st %r0 -> [fp]	
0x1030	ret	

0xfff434	0x0
0xfff438	0x6
0xfff43c	0xfff448
0xfff440	0x1028
0xfff444	0x5
0xfff448	0x0

```
int a(int i)
{
   int j;

   j = i+1;
   return j;
}

main()
{
   int i;

   i = a(5);
}
```

Prodedures with local parameters

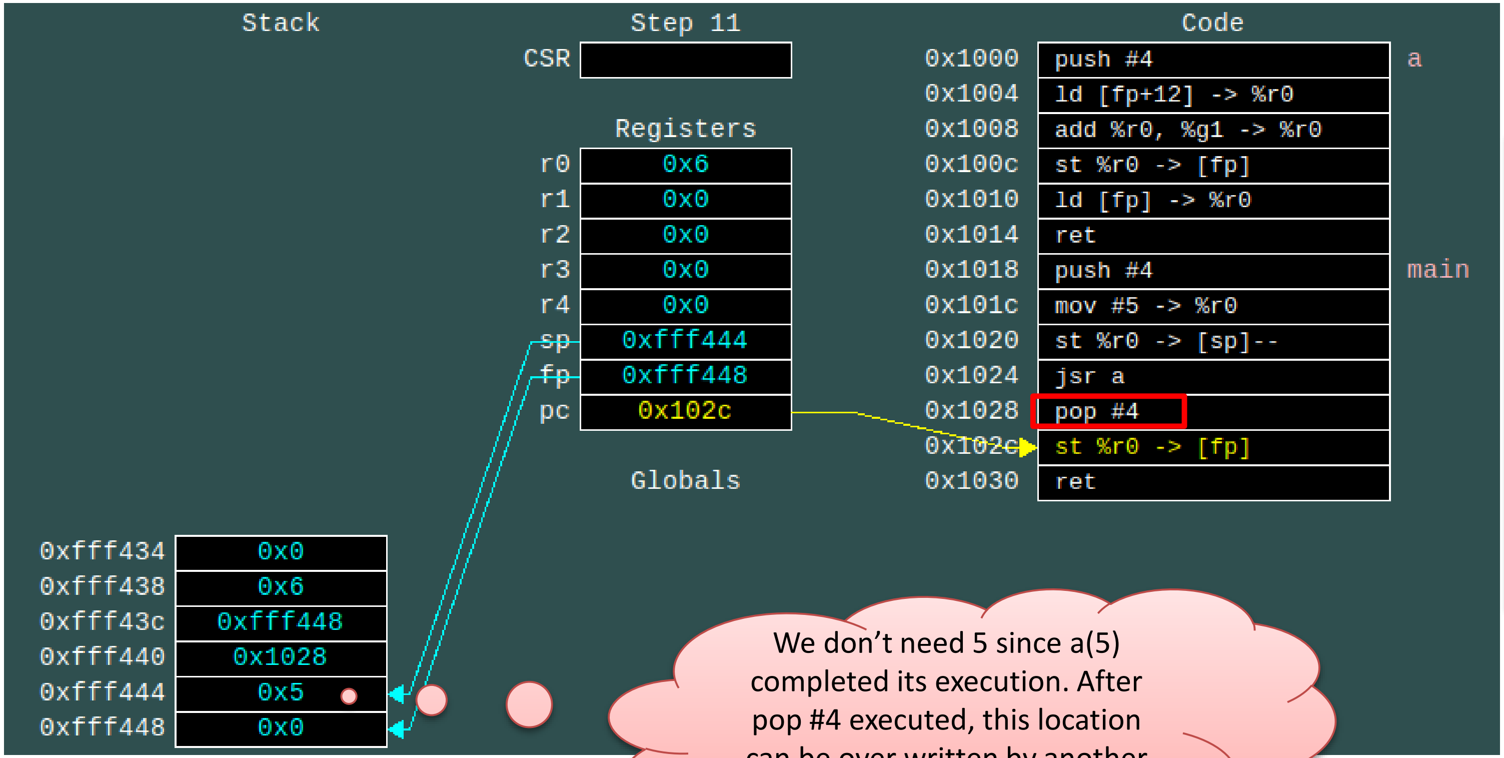


```
Stack                    Step 10                        Code
                    CSR  [        ]         0x1000  push #4                 a
                                            0x1004  ld [fp+12] -> %r0
                         Registers          0x1008  add %r0, %g1 -> %r0
                    r0   0x6                0x100c  st %r0 -> [fp]
                    r1   0x0                0x1010  ld [fp] -> %r0
                    r2   0x0                0x1014  ret
                    r3   0x0                0x1018  push #4                 main
                    r4   0x0                0x101c  mov #5 -> %r0
                    sp   0xfff440           0x1020  st %r0 -> [sp]--
                    fp   0xfff448           0x1024  jsr a
                    pc   0x1028             0x1028  pop #4
                                            0x102c  st %r0 -> [fp]
                         Globals            0x1030  ret

0xfff434  0x0
0xfff438  0x6
0xfff43c  0xfff448
0xfff440  0x1028
0xfff444  0x5
0xfff448  0x0
```

```
int a(int i)
{
   int j;

   j = i+1;
   return j;
}

main()
{
   int i;

   i = a(5);
}
```

After returning function, fp is set to its previous value used in main

Procedures with local parameters



Step 11

	CSR
CSR	

Registers

r0	0x6
r1	0x0
r2	0x0
r3	0x0
r4	0x0
sp	0xfff444
fp	0xfff448
pc	0x102c

Globals

Stack

0xfff434	0x0
0xfff438	0x6
0xfff43c	0xfff448
0xfff440	0x1028
0xfff444	0x5
0xfff448	0x0

Code

0x1000	push #4	a
0x1004	ld [fp+12] -> %r0	
0x1008	add %r0, %g1 -> %r0	
0x100c	st %r0 -> [fp]	
0x1010	ld [fp] -> %r0	
0x1014	ret	
0x1018	push #4	main
0x101c	mov #5 -> %r0	
0x1020	st %r0 -> [sp]--	
0x1024	jsr a	
0x1028	pop #4	
0x102c	st %r0 -> [fp]	
0x1030	ret	

We don't need 5 since a(5) completed its execution. After pop #4 executed, this location can be over written by another value.

```
int a(int i)
{
  int j;

  j = i+1;
  return j;
}

main()
{
  int i;

  i = a(5);
}
```

Procedures with local parameters

C code

```
int a(int i, int j)
{
  int k;

  i++;
  j -= 2;
  k = i * j;
  return k;
}

int main()
{
  int i, j, k;

  i = 3;
  j = 4;
  k = a(j+1, i);
  return 0;
}
```

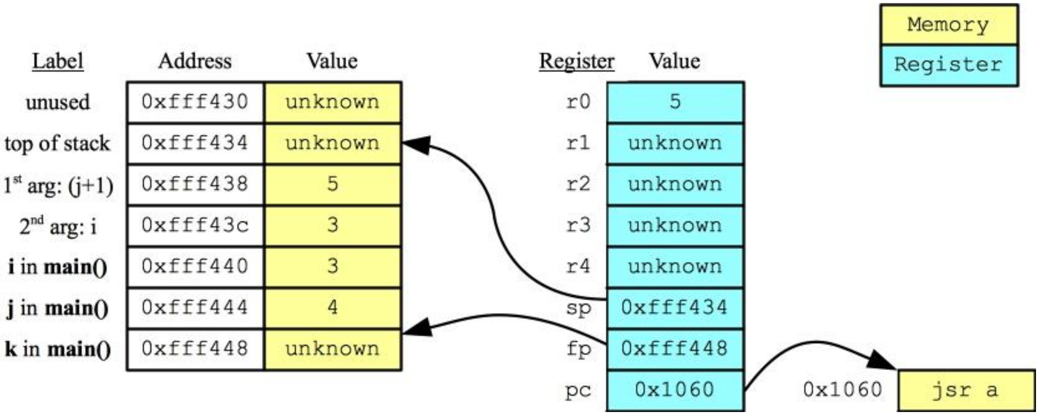
Assembly code

```
a:
    push #4     / Allocate k, which will be [fp]
    ld [fp+12] -> %r0     / i++
    add %r0, %g1 -> %r0
    st %r0 -> [fp+12]

    ld [fp+16] -> %r0    / j -= 2
    mov #2 -> %r1
    sub %r0, %r1 -> %r0
    st %r0 -> [fp+16]

    ld [fp+12] -> %r0        / k = i * j
    ld [fp+16] -> %r1
    mul %r0, %r1 -> %r0
    st %r0 -> [fp]

    ld [fp] -> %r0    / return k
    ret

main:
    push #12 / Allocate i, j, k.
    / i is [fp-8], j is [fp-4], k is [fp]

    mov #3 -> %r0            / i = 3
    st %r0 -> [fp-8]
    mov #4 -> %r0            / j = 4
    st %r0 -> [fp-4]

    ld [fp-8] -> %r0         / Push i onto the stack
    st %r0 -> [sp]--

    ld [fp-4] -> %r0         / Push j+1 onto the stack
    add %r0, %g1 -> %r0
    st %r0 -> [sp]--

    jsr a                    / Call a(), then pop the arguments
    pop #8

    st %r0 -> [fp]          / Put the return value into k

    mov #0 -> %r0           / Return 0
    ret
```

Procedures with local parameters

Before executing *jsr a* line

```
int main()
{
  int i, j, k;

  i = 3;
  j = 4;
  k = a(j+1, i);
  return 0;
}
```



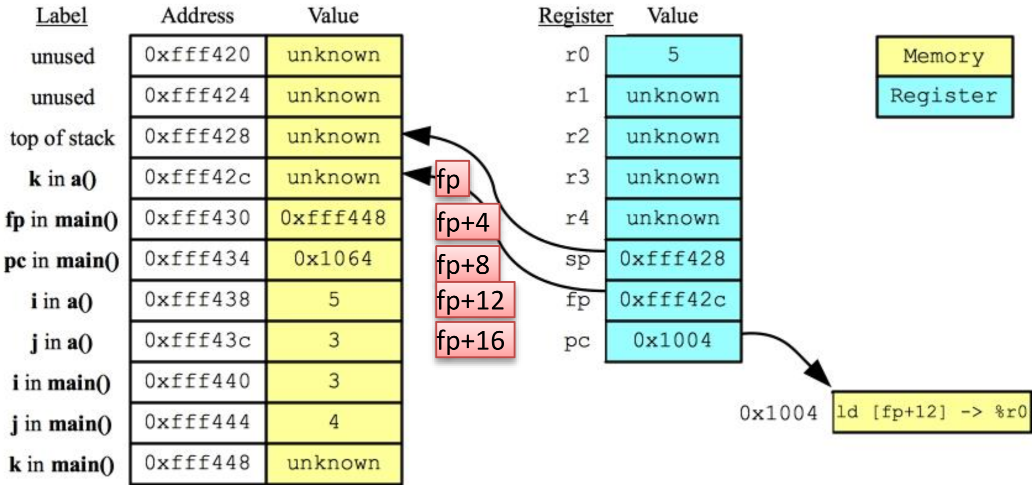
Label	Address	Value
unused	0xfff430	unknown
top of stack	0xfff434	unknown
1st arg: (j+1)	0xfff438	5
2nd arg: i	0xfff43c	3
i in **main()**	0xfff440	3
j in **main()**	0xfff444	4
k in **main()**	0xfff448	unknown

Register	Value
r0	5
r1	unknown
r2	unknown
r3	unknown
r4	unknown
sp	0xfff434
fp	0xfff448
pc	0x1060

```
main:
    push #12 / Allocate i, j, k.
    / i is [fp-8], j is [fp-4], k is [fp]

    mov #3 -> %r0            / i = 3
    st %r0 -> [fp-8]
    mov #4 -> %r0            / j = 4
    st %r0 -> [fp-4]

    ld [fp-8] -> %r0         / Push i onto the stack
    st %r0 -> [sp]--

    ld [fp-4] -> %r0         / Push j+1 onto the stack
    add %r0, %g1 -> %r0
    st %r0 -> [sp]--

    jsr a                / Call a(), then pop the arguments
    pop #8

    st %r0 -> [fp]          / Put the return value into k

    mov #0 -> %r0           / Return 0
    ret
```

Procedures with local parameters

Accessing local parameter



Label	Address	Value
unused	0xfff420	unknown
unused	0xfff424	unknown
top of stack	0xfff428	unknown
k in **a()**	0xfff42c	unknown
fp in **main()**	0xfff430	0xfff448
pc in **main()**	0xfff434	0x1064
i in **a()**	0xfff438	5
j in **a()**	0xfff43c	3
i in **main()**	0xfff440	3
j in **main()**	0xfff444	4
k in **main()**	0xfff448	unknown

Register	Value
r0	5
r1	unknown
r2	unknown
r3	unknown
r4	unknown
sp	0xfff428
fp	0xfff42c
pc	0x1004

```
int a(int i, int j)
{
  int k;

  i++;
  j -= 2;
  k = i * j;
  return k;
}
```

```
a:
    push #4     / Allocate k, which will be [fp]
    ld [fp+12] -> %r0     / i++
    add %r0, %g1 -> %r0
    st %r0 -> [fp+12]

    ld [fp+16] -> %r0    / j -= 2
    mov #2 -> %r1
    sub %r0, %r1 -> %r0
    st %r0 -> [fp+16]

    ld [fp+12] -> %r0        / k = i * j
    ld [fp+16] -> %r1
    mul %r0, %r1 -> %r0
    st %r0 -> [fp]

    ld [fp] -> %r0    / return k
    ret
```

Register Spilling

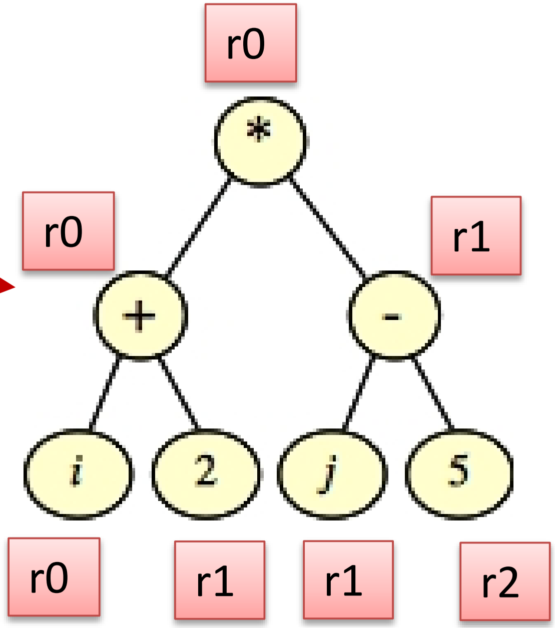
- One important thing that has to be decided is whether a procedure may use a register without worrying about its current value (like a() does with r0), or whether a procedure should first save the register on the stack before using it.
- This matters, because suppose for example, that the main routine uses register r3, then calls "jsr a", and afterwards expects r3 to have the same value.
- Then a() and any procedures that a() calls must make sure not to use r3, or to save r3's value before using it, and restore it when its done.

Register Spilling

- One important thing that has to be decided is whether a procedure may use a register without worrying about its current value (like a() does with r0), or whether a procedure should first save the register on the stack before using it.
- This matters, because suppose for example, that the main routine uses register **r3**, then calls "**jsr a**", and afterwards expects **r3** to have the same value.
- Then **a()** and any procedures that **a()** calls must make sure not to use **r3**, or to save **r3**'s value before using it, and restore it when its done
- The act of saving a register's value before the body of a procedure call and restoring it afterwards is called *spilling*.
- Different machines and compilers handle spilling in different ways. For example, older CISC architectures sometimes had a spill-mask that would be part of a procedure call.
- This specifies which registers should be spilled, and the machine actually did the spilling for you.
- What we do on our machine is a typical spilling solution: Procedures can use **r0** and **r1** without worrying about their values. However, registers **r2** through **r4** must be spilled if a procedure uses them

Register Spilling

```
int a(int i, int j)
{
  int k;

  k = (i+2)*(j-5);

  return k;
}

main()
{
  int i;

  i = a(44, 22);

}
```



- In order to evaluate the tree, you need to do a postorder traversal (or, if you think of the edges are pointing upward, you need to do a topological sorting of the tree).
- Arithmetic has to be done on a register-by-register basis, so each of those nodes must be in a register.
- You (the compiler) must figure out an ordering of instructions that is legal, and then an assignment of nodes to registers so that you don't reuse registers unless you can be sure that you don't need their values any more.
- For example, in the above expression, suppose you do the (i+2) calculation first and hold the result in **r0**.
- Then you can't use **r0** to calculate (j-5). For that reason, you are going to have to use **r2**, and because you are using **r2**, you'll have to spill it onto the stack. I do this at the beginning of a procedure. Then at the end, I "unspill" it by reading it back from the stack.

Register Spilling

C kodu	Assembly kod

C kodu

```
int a(int i, int j)
{

  int k;


  k = (i+2)*(j-5);


  return k;

}

main()
{
  int i;

  i = a(44, 22);

}
```

Assembly kod

```
a:
    push #4              / Allocate k
    st %r2 -> [sp]--     / Spill r2

    ld [fp+12] -> %r0
    mov #2 -> %r1
    add %r0, %r1 -> %r0  / Calculate (i+2) and put the result in r0

    ld [fp+16] -> %r1
    mov #5 -> %r2
    sub %r1, %r2 -> %r1 / Calculate (j-5) and put the result in r1

    mul %r0, %r1 -> %r0
    st %r0 -> [fp]      / Do k = r0 * r1
    ld [fp] -> %r0

    ld ++[sp] -> %r2    / Unspill r2
    ret

main:
    push #4             / Allocate i

    mov #22 -> %r0      / Push arguments onto
    st %r0 -> [sp]--    / the stack in reverse order

    mov #44 -> %r0
    st %r0 -> [sp]--

    jsr a
    pop #8              / Always pop the arguments off the stack after jsr
    st %r0 -> [fp]
    ret
```

Register Spilling

C kodu

```c
int a(int i, int j)
{
  int k;

  k = (i+2)*(j-5);
  return k;
}



int main()
{
  int i;

  i =(a(10,20) + a(30,40));
}
```

Assembly kod

```
a:
    push #4                 / Allocate k
    st %r2 -> [sp]--        / Spill r2

    ld [fp+12] -> %r0
    mov #2 -> %r1
    add %r0, %r1 -> %r0     / Calculate (i+2) and put the result in r0

    ld [fp+16] -> %r1
    mov #5 -> %r2
    sub %r1, %r2 -> %r1     / Calculate (j-5) and put the result in r1

    mul %r0, %r1 -> %r0
    st %r0 -> [fp]          / Do k = r0 * r1

    ld [fp] -> %r0
    ld ++[sp] -> %r2        / Unspill r2
    ret

main:

    push #4                 / Allocate i
    st %r2 -> [sp]--        / Spill r2

    mov #20 -> %r0          / Call a(10, 20) and store the result in r2
    st %r0 -> [sp]--
    mov #10 -> %r0
    st %r0 -> [sp]--
    jsr a
    pop #8
    mov %r0 -> %r2

    mov #40 -> %r0          / Call a(30, 40) and add the result to r2
    st %r0 -> [sp]--
    mov #30 -> %r0
    st %r0 -> [sp]--
    jsr a
    pop #8
    add %r0, %r2 -> %r0
    st %r0 -> [fp]

    ld ++[sp] -> %r2        / Unspill r2
    ret
```

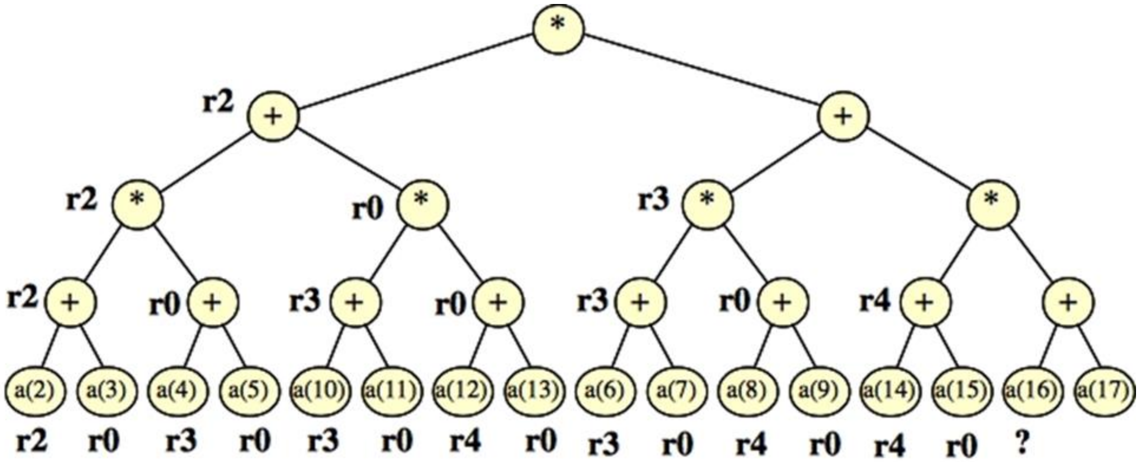
Register Spilling

What do you do when you run out of registers?

```
int a(int i)
{
  return i+5;
}

int main()
{
  int i;

  i = ( (a(2)+a(3)) * (a(4)+a(5)) + (a(10)+a(11)) * (a(12)+a(13)) ) *
      ( (a(6)+a(7)) * (a(8)+a(9)) + (a(14)+a(15)) * (a(16)+a(17)) );
}
```

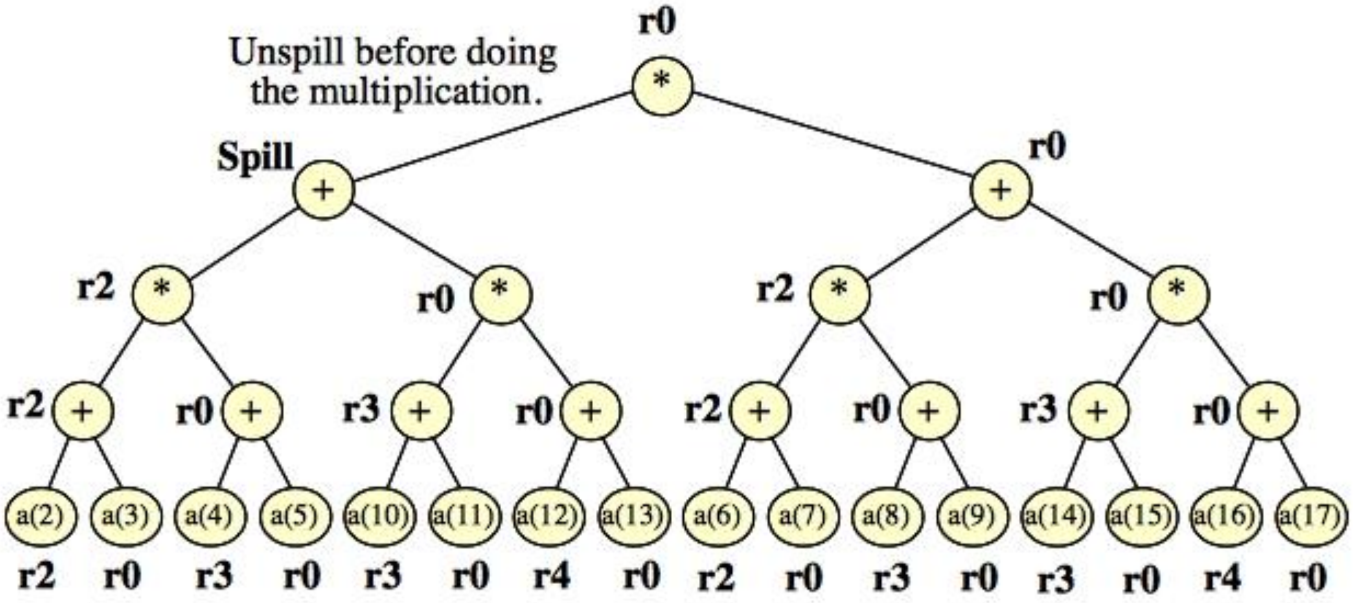
```
a:
   ld [fp+12] -> %r0
   mov #5 -> %r1
   add %r0, %r1 -> %r0
   ret
```

- You can see I've labeled it with the registers that you can use if you do the calculation in post-order, from left to right. You'll see that we've run out of registers!



Register Spilling

- Below, I show how you handle that -- you spill the intermediate value shown as "Spill".
- That allows you to use **r2** again, and you no longer run out of registers. Before you do the last multiplication, you unspill the value into a register:



Unspill before doing the multiplication.

Register Spilling

```
a:
   ld [fp+12] -> %r0
   mov #5 -> %r1
   add %r0, %r1 -> %r0
   ret
main:
   push #4
   st %r2 -> [sp]--      / You have to spill r2, r3, and r4
   st %r3 -> [sp]--
   st %r4 -> [sp]--

   mov #2 -> %r0         / a(2)+a(3)
   st %r0 -> [sp]--
   jsr a
   pop #4
   mov %r0 -> %r2

   mov #3 -> %r0
   st %r0 -> [sp]--
   jsr a
   pop #4
   add %r0, %r2 -> %r2

   mov #4 -> %r0         / a(4)+a(5)
   st %r0 -> [sp]--
   jsr a
   pop #4
   mov %r0 -> %r3

   mov #5 -> %r0
   st %r0 -> [sp]--
   jsr a
   pop #4
   add %r0, %r3 -> %r0

   mul %r2, %r0 -> %r2   / Multiplication

   mov #10 -> %r0         / a(10)+a(11)
   st %r0 -> [sp]--
   jsr a
   pop #4
   mov %r0 -> %r3
```

```
   mov #11 -> %r0
   st %r0 -> [sp]--
   jsr a
   pop #4
   add %r0, %r3 -> %r3

   mov #12 -> %r0        / a(12)+a(13)
   st %r0 -> [sp]--
   jsr a
   pop #4
   mov %r0 -> %r4

   mov #13 -> %r0
   st %r0 -> [sp]--
   jsr a
   pop #4
   add %r0, %r4 -> %r0
   mul %r3, %r0 -> %r0   / Multiplication
   add %r2, %r0 -> %r0   / then Addition, then spill
   st %r0 -> [sp]--          / then spill

   mov #6 -> %r0           / a(6)+a(7)
   st %r0 -> [sp]--
   jsr a
   pop #4
   mov %r0 -> %r2

   mov #7 -> %r0
   st %r0 -> [sp]--
   jsr a
   pop #4
   add %r0, %r2 -> %r2

   mov #8 -> %r0             / a(8)+a(9)
   st %r0 -> [sp]--
   jsr a
   pop #4
   mov %r0 -> %r3
```

```
   mov #9 -> %r0
   st %r0 -> [sp]--
   jsr a
   pop #4
   add %r0, %r3 -> %r0

  mul %r2, %r0 -> %r2    / Multiplication

  mov #14 -> %r0         / a(14)+a(15)
   st %r0 -> [sp]--
   jsr a
   pop #4
   mov %r0 -> %r3

   mov #15 -> %r0
   st %r0 -> [sp]--
   jsr a
   pop #4
   add %r0, %r3 -> %r3

   mov #16 -> %r0        / a(16)+a(17)
   st %r0 -> [sp]--
   jsr a
   pop #4
   mov %r0 -> %r4

   mov #17 -> %r0
   st %r0 -> [sp]--
   jsr a
   pop #4
   add %r0, %r4 -> %r0

   mul %r3, %r0 -> %r0   / Multiplication
   add %r2, %r0 -> %r0   / then addition
   ld ++[sp] -> %r1         / then unspill
   mul %r0, %r1 -> %r0

   st %r0 -> [fp]

   ld ++[sp] -> %r4     / Unspill before returning
   ld ++[sp] -> %r3
   ld ++[sp] -> %r2
   ret
```