Assembler 3

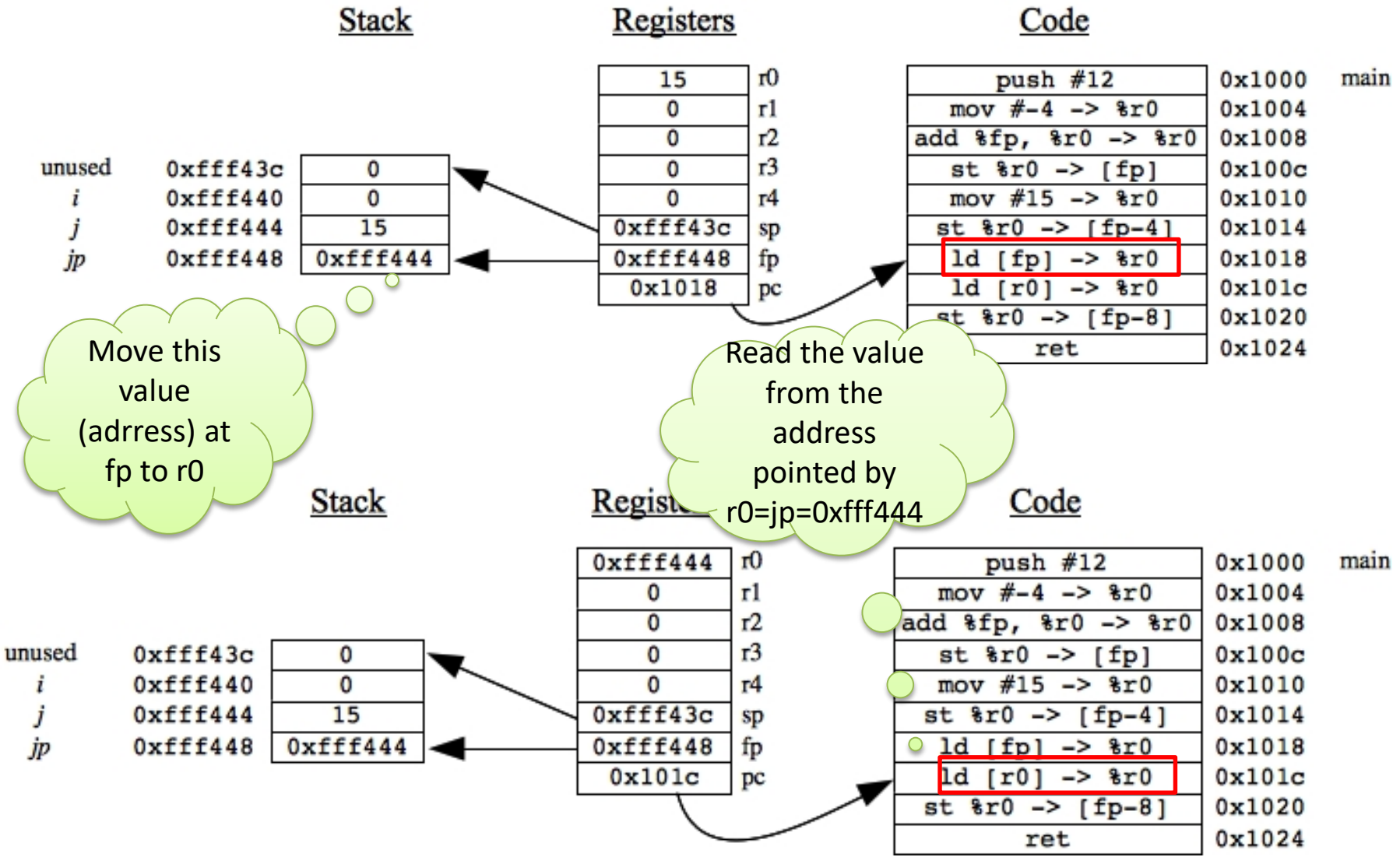
Pointers

C code	Assembly code

`main()`
`{`
`  int i, j, *jp;`
 	
`  jp = &j;`
`  j = 15;`
`  i = *jp;`
`}`
	`main:`
	`push #12`
	`mov #-4 -> %r0  / jp = &j.`
	`add %fp, %r0 -> %r0`
	`st %r0 -> [fp]`
	
	`mov #15 -> %r0  / j = 15`
	`st %r0 -> [fp-4]`
	
	`ld [fp] -> %r0  / i = *jp`
	`ld [r0] -> %r0`
	`st %r0 -> [fp-8]`
	`ret`

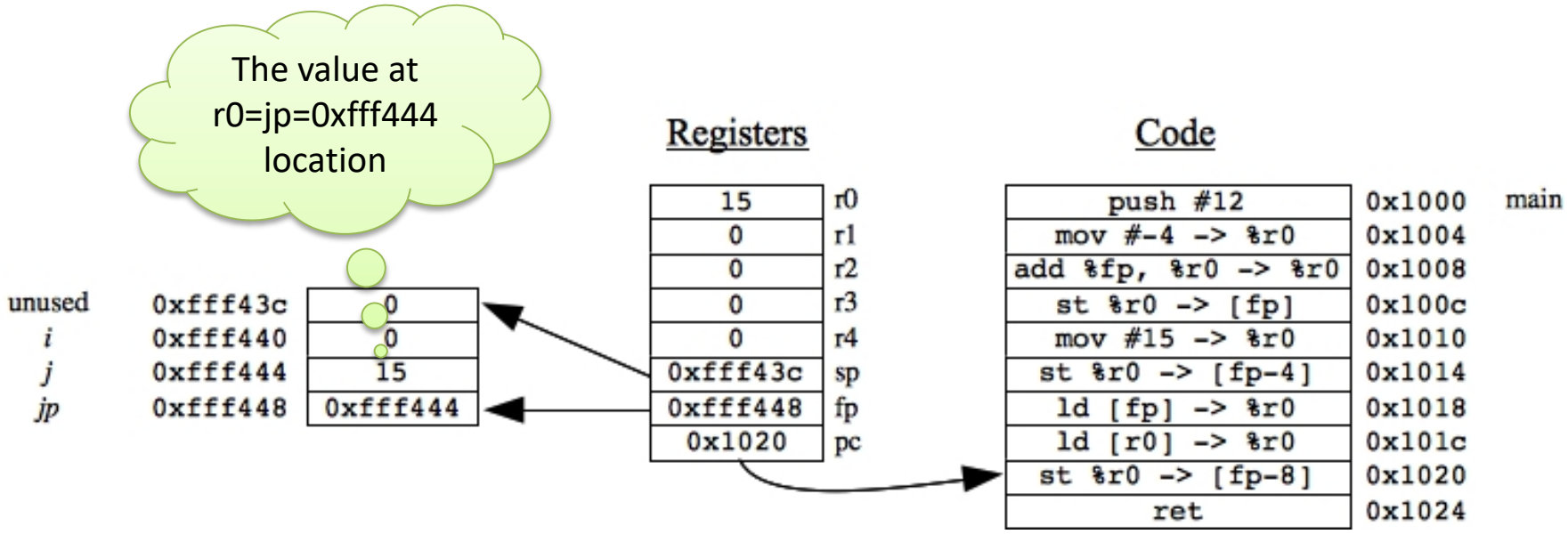
Pointers

Execution of program
for i = *jp



Pointers

Execution of program
for i = *jp



Procedure with pointers

C kodu	Assembly kod

`int a(int *p)`
`{`
`  return *p;`
`}`
 	
`main()`
`{`
`  int i, j;`
 	
`  j = 15;`
`  i = a(&j);`
`}`
	`a:`
	`  ld [fp+12] -> %r0  / get p's value`
	`  ld [r0] -> %r0  / dereference it`
	`  ret`
	
	`main:`
	`  push #8`
	
	`  mov #15 -> %r0  / j = 15`
	`  st %r0 -> [fp]`
	
	`  st %fp -> [sp]--  / push &j on the stack`
	`  jsr a  / and call a()`
	`  pop #4`
	`  st %r0 -> [fp-4]`
	
	`  ret`

Array operations

Array operations

C code	Assembly code

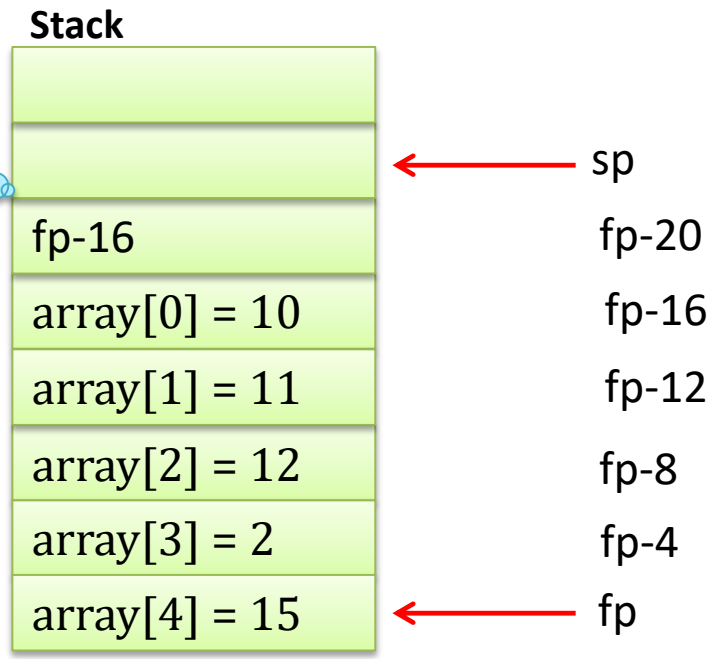
`int main()`
`{`
`  int array[5];`
 	
`  array[0] = 10;`
`  array[1] = 11;`
`  array[2] = 12;`
`  array[3] = 2;`
`  array[4] = 15;`
 	
`  a(array);`
`}`
	`main:`
	`  push #20`
	
	`  mov #10 -> %r0  / Store the values of array`
	`  st %r0 -> [fp-16]`
	`  mov #11 -> %r0`
	`  st %r0 -> [fp-12]`
	`  mov #12 -> %r0`
	`  st %r0 -> [fp-8]`
	`  mov #2 -> %r0`
	`  st %r0 -> [fp-4]`
	`  mov #15 -> %r0`
	`  st %r0 -> [fp]`
	
	`  mov #-16 -> %r0  / Push array onto the stack`
	`  add %fp, %r0 -> %r0`
	`  st %r0 -> [sp]--`
	`  jsr a  / call a`
	`  pop #4`
	`  ret`

Array operations

```
main()
{
  int array[5];
  array[0] = 10;
  array[1] = 11;
  array[2] = 12;
  array[3] = 2;
  array[4] = 15;

  a(array);
}
```

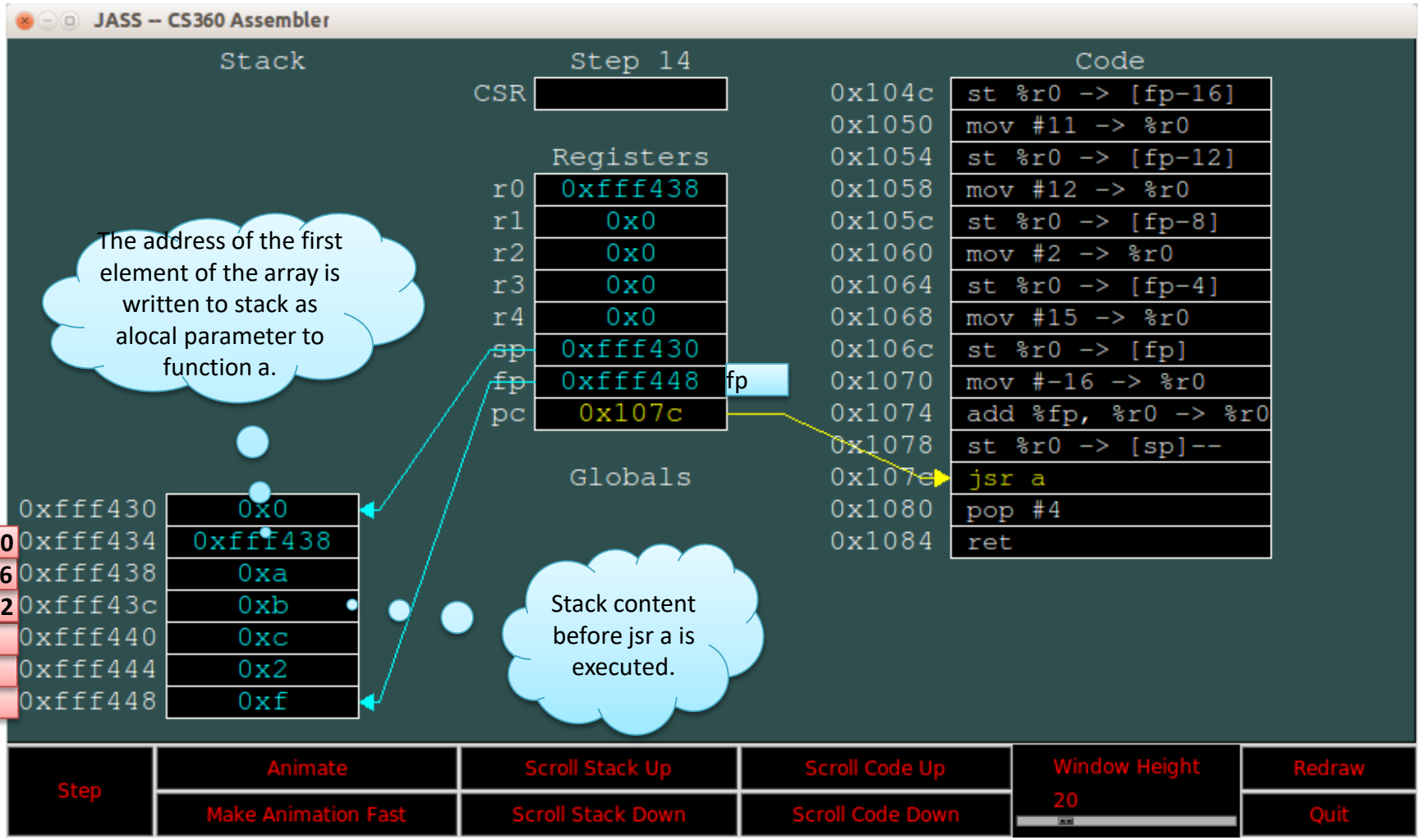
```
main:
    push #20

    mov #10 -> %r0
    st %r0 -> [fp-16]
    mov #11 -> %r0
    st %r0 -> [fp-12]
    mov #12 -> %r0
    st %r0 -> [fp-8]
    mov #2 -> %r0
    st %r0 -> [fp-4]
    mov #15 -> %r0
    st %r0 -> [fp]

    mov #-16 -> %r0
    add %fp, %r0 -> %r0
    st %r0 -> [sp]--

    jsr a                    / call a
    pop #4
    ret
```

Stack before executing **jsr a**

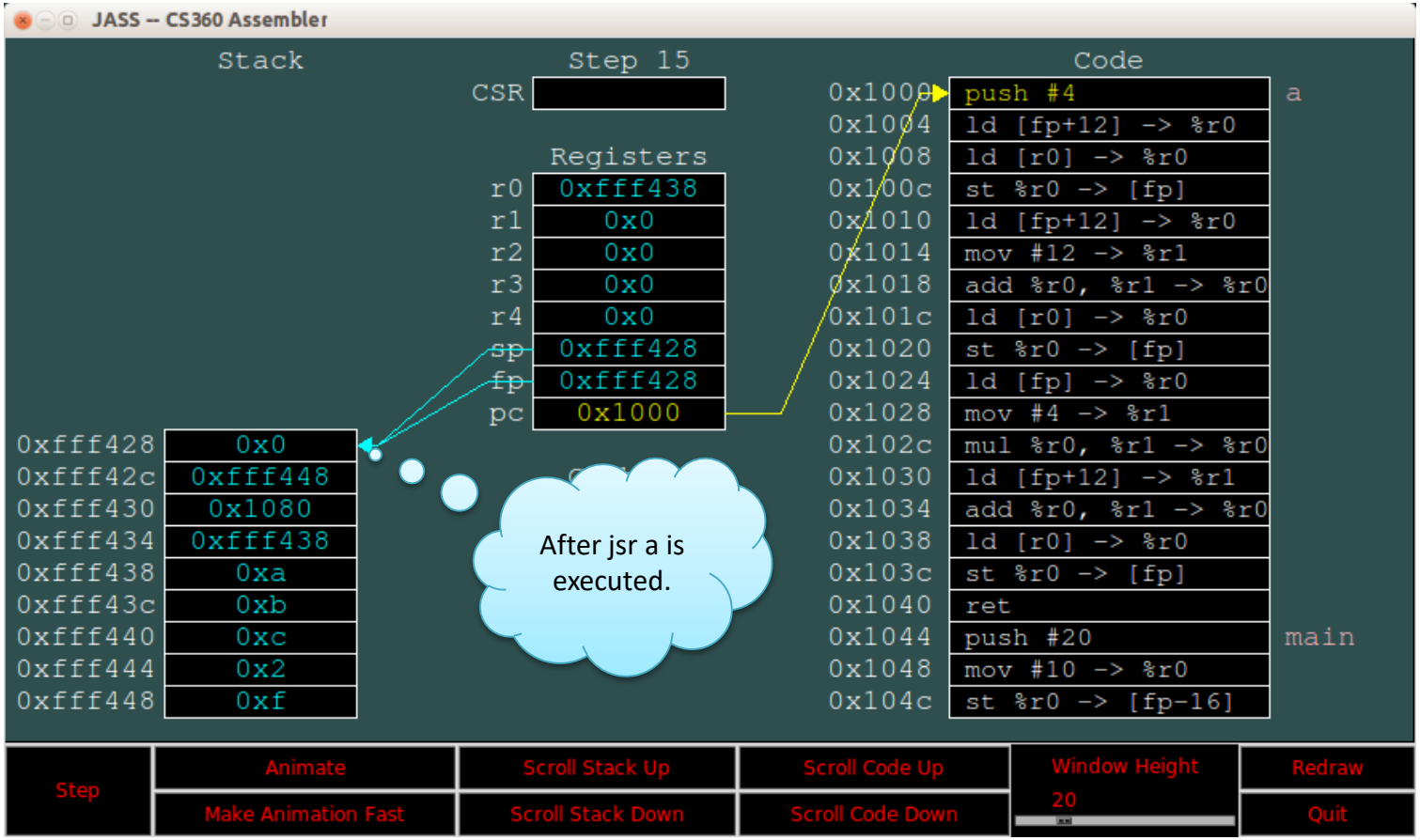


Dizinin başlangıç adresi fp-16 adres değerini, array[0] elemanının bir üstüne yaz. Bunun için fp ile -16 toplanıyor ve sonuç stack pointer ile gösterilen fp-20 adresine kaydedildikten sonra stack pointer azaltılarak sıradaki boş yeri gösteriyor.

Array operations



Array operations



Array operations

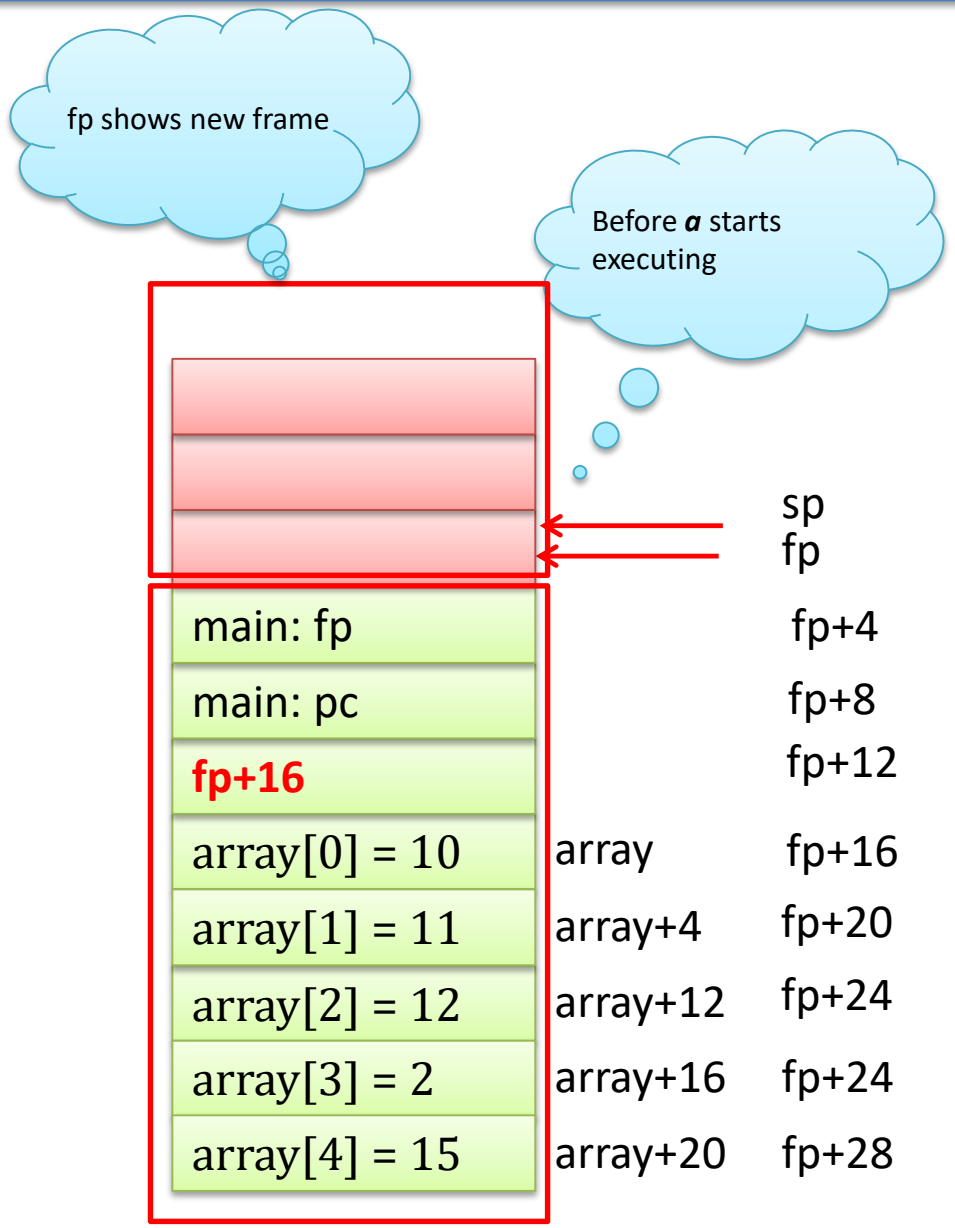
C code	Assembly code

`void a(int *p)`
`{`
`  int i;`
 	
`  i = p[0];`
`  i = p[3];`
`  i = p[i];`
`}`
	`a:`
	`  push #4`
	`  ld [fp+12] -> %r0  / i = p[0]`
	`  ld [r0] -> %r0`
	`  st %r0 -> [fp]`
	
	`  ld [fp+12] -> %r0  / i = p[3]`
	`  mov #12 -> %r1`
	`  add %r0, %r1 -> %r0`
	`  ld [r0] -> %r0`
	`  st %r0 -> [fp]`
	
	`  ld [fp] -> %r0  / i = p[i]`
	`  mov #4 -> %r1`
	`  mul %r0, %r1 -> %r0`
	`  ld [fp+12] -> %r1`
	`  add %r0, %r1 -> %r0`
	`  ld [r0] -> %r0`
	`  st %r0 -> [fp]`
	
	`  ret`

Array operations

```
a(int *p)
{
  int i;

  i = p[0];
  i = p[3];
  i = p[i];
}
```

```
a:
    push #4

    ld [fp+12] -> %r0
    ld [r0] -> %r0
    st %r0 -> [fp]

    ld [fp+12] -> %r0
    mov #12 -> %r1
    add %r0, %r1 -> %r0
    ld [r0] -> %r0
    st %r0 -> [fp]

    ld [fp] -> %r0
    mov #4 -> %r1
    mul %r0, %r1 -> %r0
    ld [fp+12] -> %r1
    add %r0, %r1 -> %r0
    ld [r0] -> %r0
    st %r0 -> [fp]
    ret
```

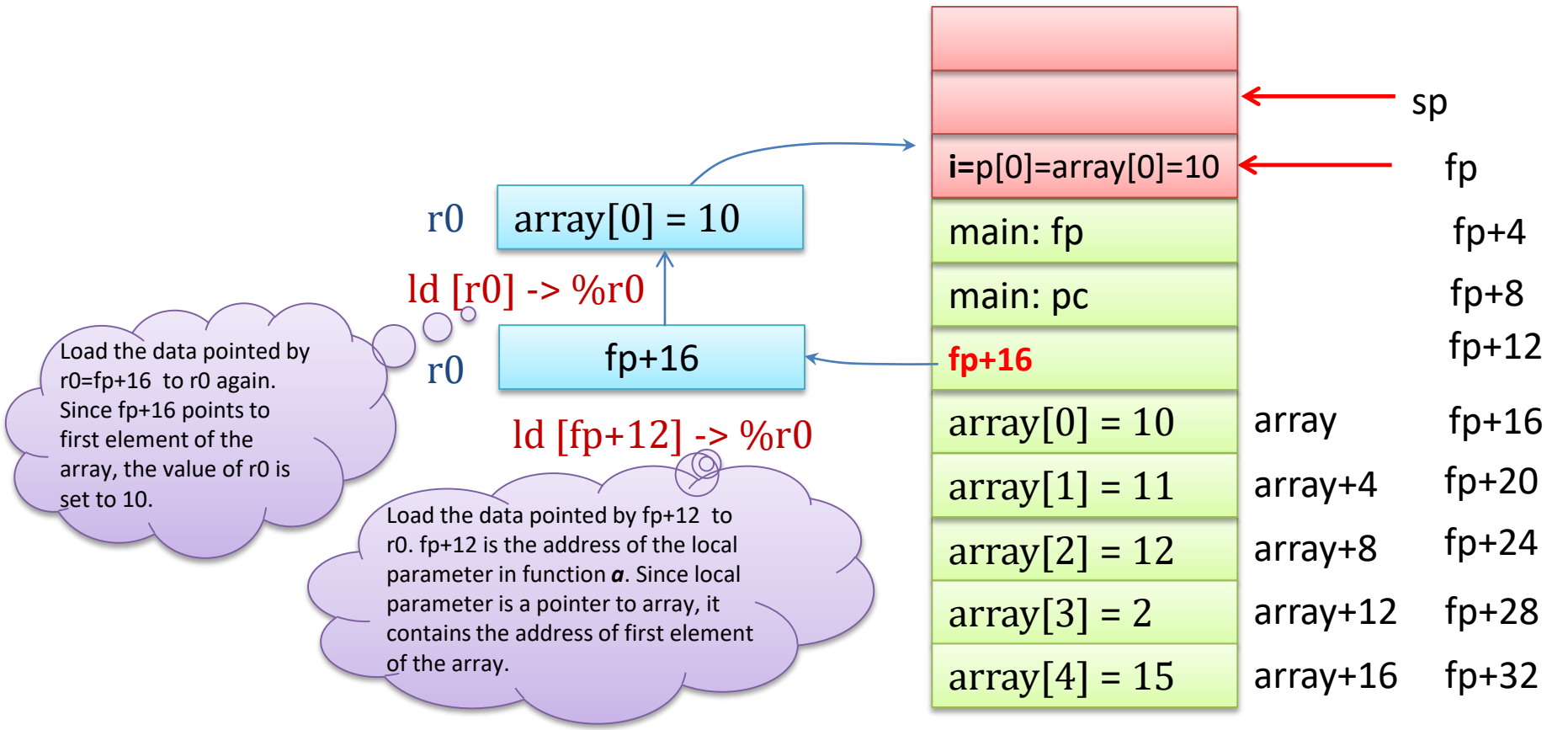


Array operations

```
a:
    push #4

    ld [fp+12] -> %r0
    ld [r0] -> %r0
    st %r0 -> [fp]
```

```
a(int *p)
{
  int i;

  i = p[0];
  i = p[3];
  i = p[i];
}
```

Stack after executing;
int i;
i = p[0];

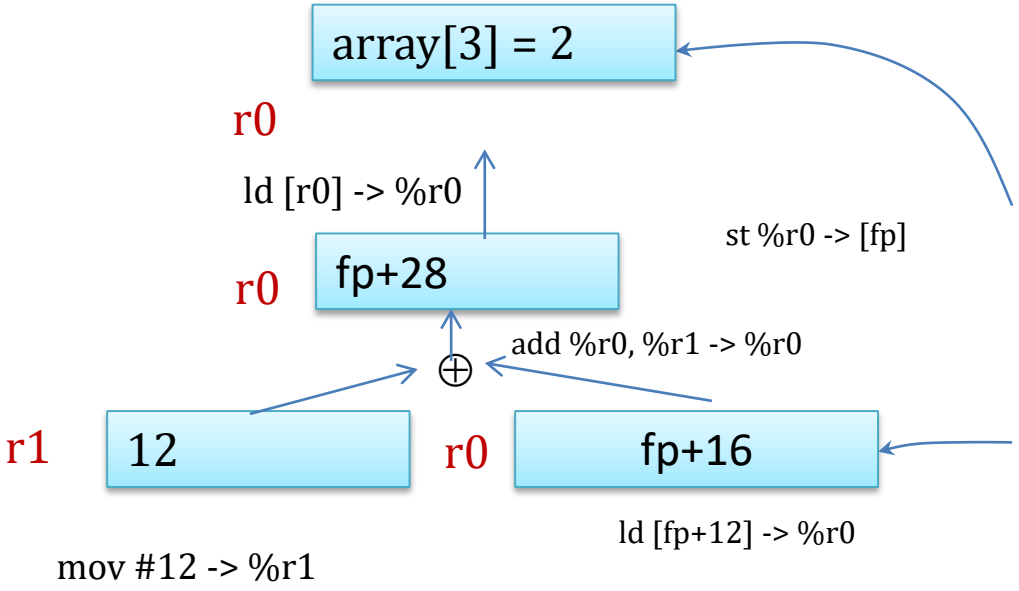


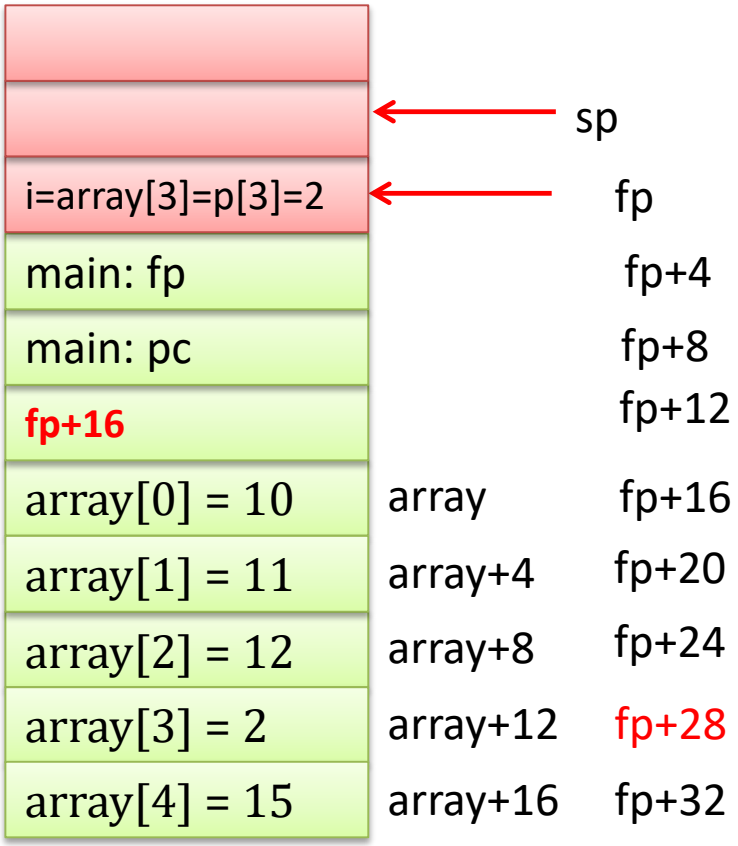
Array operations

```
ld [fp+12] -> %r0
mov #12 -> %r1
add %r0, %r1 -> %r0
ld [r0] -> %r0
st %r0 -> [fp]
```

```
a(int *p)
{
  int i;

  i = p[0];
  i = p[3];
  i = p[i];
}
```

Stack after executing; i = p[3];

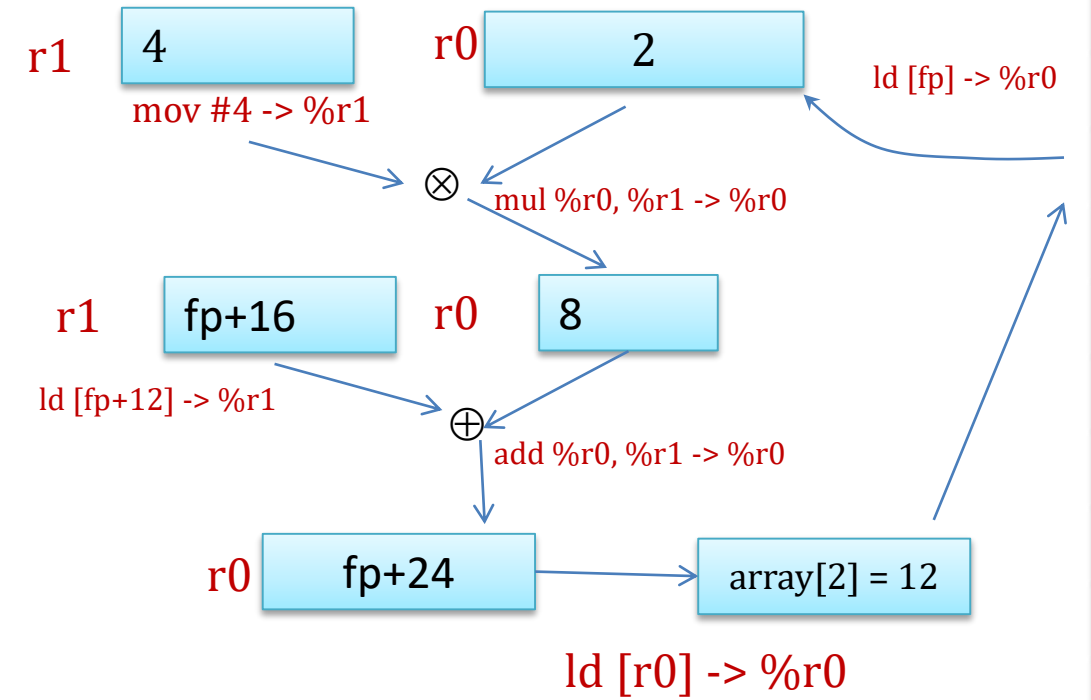


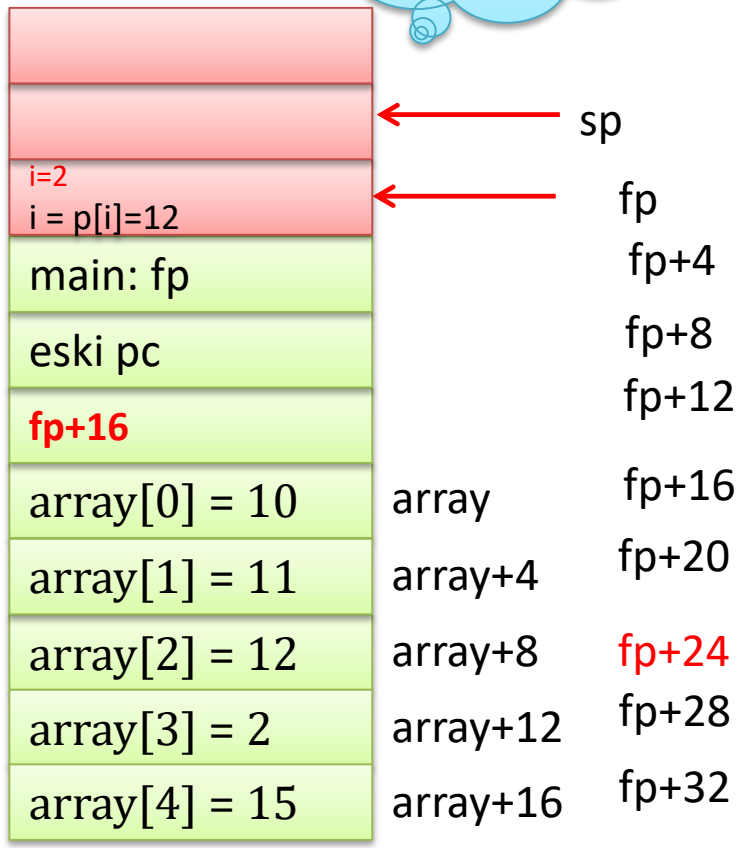


Array operations

```
ld [fp] -> %r0
mov #4 -> %r1
mul %r0, %r1 -> %r0
ld [fp+12] -> %r1
add %r0, %r1 -> %r0
ld [r0] -> %r0
st %r0 -> [fp]
```

```
a(int *p)
{
  int i;

  i = p[0];
  i = p[3];
  i = p[i];
}
```





Uninitialized Locals

C code

```
main()
{
  int x;

  a(5, 6, 7, 8);
  x = b(2);
}
```

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

  m = i + j + k + l;
}
```

```
int b(int i)
{
  int p[5];

  return p[i];
}
```

we don't initialize **p** and we simply return **p[i]**. It returns a value from the specified location.

Assembly code

```
main:
  push #4

  mov #8 -> %r0
  st %r0 -> [sp]--
  mov #7 -> %r0
  st %r0 -> [sp]--
  mov #6 -> %r0
  st %r0 -> [sp]--
  mov #5 -> %r0
  st %r0 -> [sp]--
  jsr a
  pop #16

  mov #2 -> %r0
  st %r0 -> [sp]--
  jsr b
  pop #4
  st %r0 -> [fp]
  ret
```

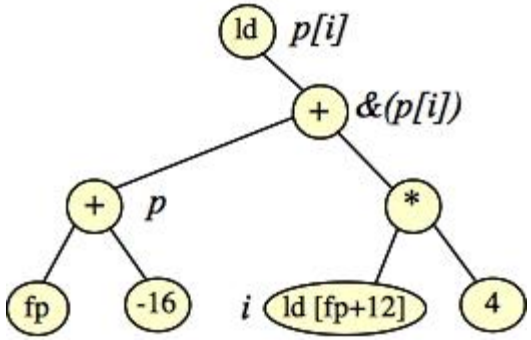
```
a:
  push #4

  ld [fp+12] -> %r0
  ld [fp+16] -> %r1
  add %r0, %r1 -> %r0
  ld [fp+20] -> %r1
  add %r0, %r1 -> %r0
  ld [fp+24] -> %r1
  add %r0, %r1 -> %r0
  st %r0 -> [fp]
  ret
```

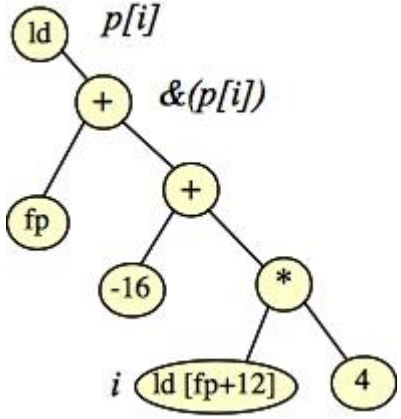
```
b:
  push #20

  ld [fp+12] -> %r0
  mov #4 -> %r1
  mul %r0, %r1 -> %r0
  mov #-16 -> %r1
  add %r0, %r1 -> %r0
  add %r0, %fp -> %r0
  ld [r0] -> %r0
  ret
```

Uninitialized Locals

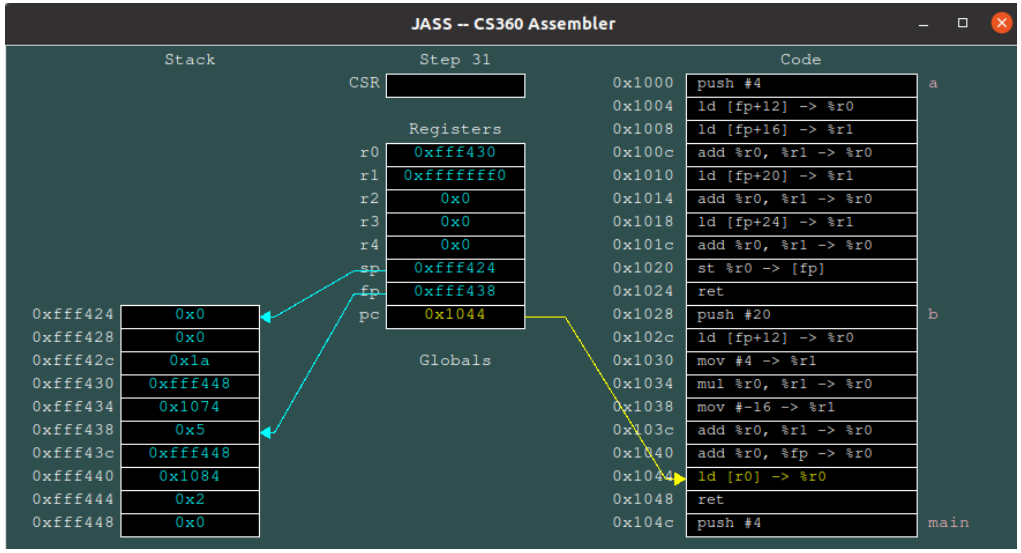
In b(), we allocate the 20 bytes for p on the stack, and we know that p points to element p[0], which is at address (fp-16). Thus, to access p[i] we need to do:

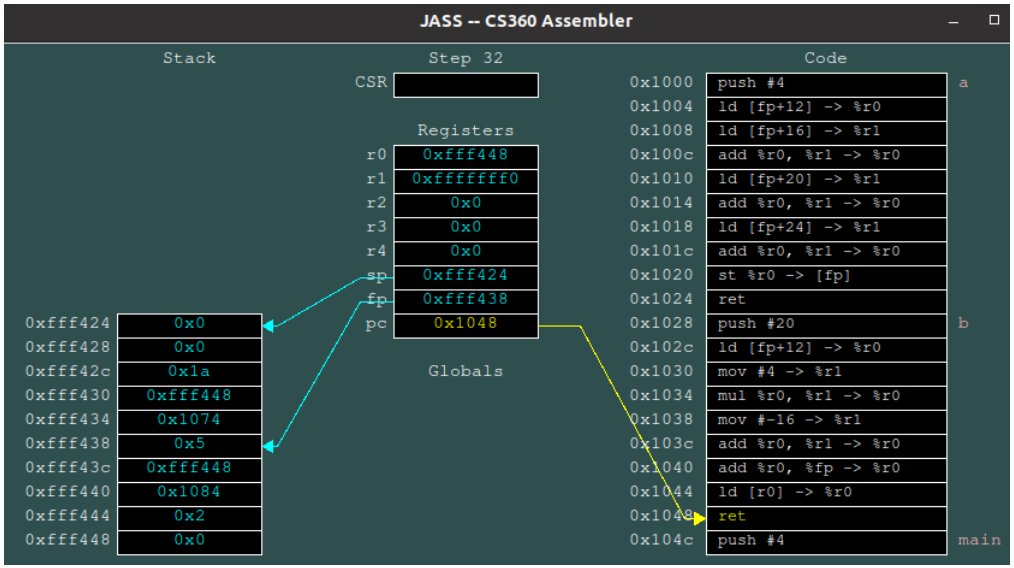
&p[i] = [(fp-16) + 4*[fp+12]]





Uninitialized Locals

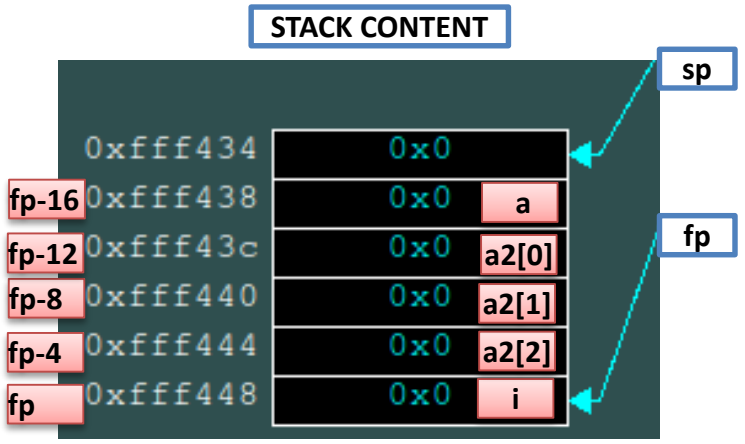




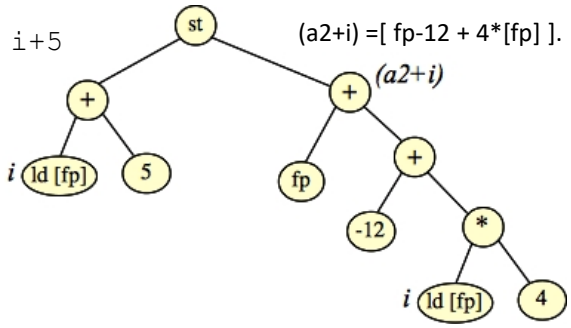
Some more practice

C code

```
main()
{
  int *a, a2[3], i;

  i = 6;
  a = &i;
  a2[1] = i+2;
  *a = 2;
  *(a2+i) = i+5;
}
```



- &i will be fp.
- a2 will be (fp-12).
- a will be [[fp-16]]. You can't do that in assembler. Instead you will load [fp-16] into a register and dereference that register.
- (a2+i) will be [fp-12 + 4*[fp]]. Remember -- that's pointer arithmetic.

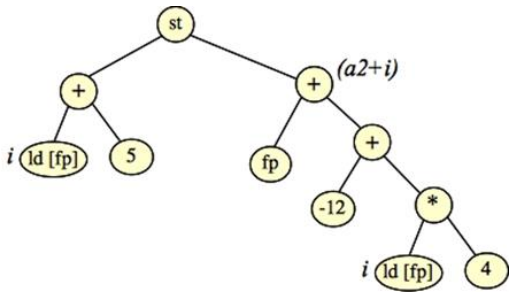


Some more practice

C code	Assembly code

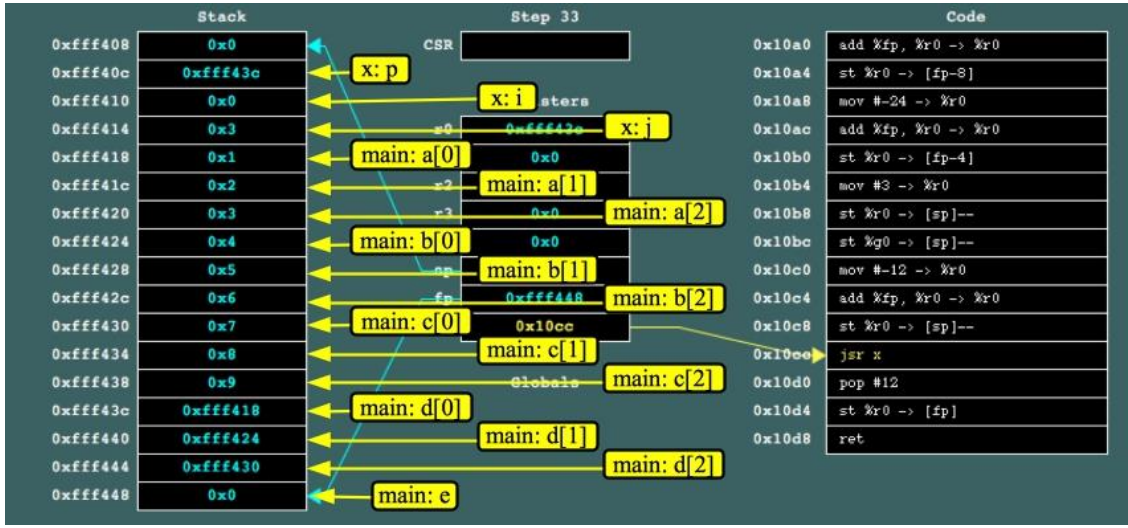
```
main()
{
  int *a, a2[3], i;

  i = 6;
  a = &i;
  a2[1] = i+2;
  *a = 2;
  *(a2+i) = i+5;
}
```

```
main:
  push #20              / Allocate locals and spill r2
  st %r2 -> [sp]--

  mov #6 -> %r0         / i = 6
  st %r0 -> [fp]

  st %fp -> [fp-16]     / a = &i

  mov #2 -> %r0         / a2[1] = i+2
  ld [fp] -> %r1
  add %r0, %r1 -> %r0
  st %r0 -> [fp-8]

  mov #2 -> %r0         / *a = 2
  ld [fp-16] -> %r1
  st %r0 -> [r1]

  ld [fp] -> %r0        / *(a+i) = i+5
  mov #5 -> %r1
  add %r0, %r1 -> %r0
  ld [fp] -> %r1
  mov #4 -> %r2
  mul %r1, %r2 -> %r1
  mov #-12 -> %r2
  add %r1, %r2 -> %r1
  add %fp, %r1 -> %r1
  st %r0 -> [r1]

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

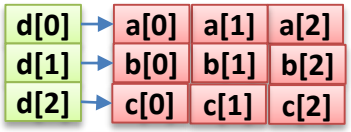


Double Indirection

```
int x(int **p, int i, int j)
{
  return p[i+2][j-2];
}

main()
{
  int a[3], b[3], c[3];
  int *d[3]; int e;

  a[0] = 1; a[1] = 2; a[2] = 3;
  b[0] = 4; b[1] = 5; b[2] = 6;
  c[0] = 7; c[1] = 8; c[2] = 9;

  //d[3][3]
  d[0] = a; d[1] = b; d[2] = c;

  e = x(d, 0, 3);
}
```



d[3][3]



d[0][0]=a[0]

d[0][1]=a[1]

...

d[2][2]=c[2]

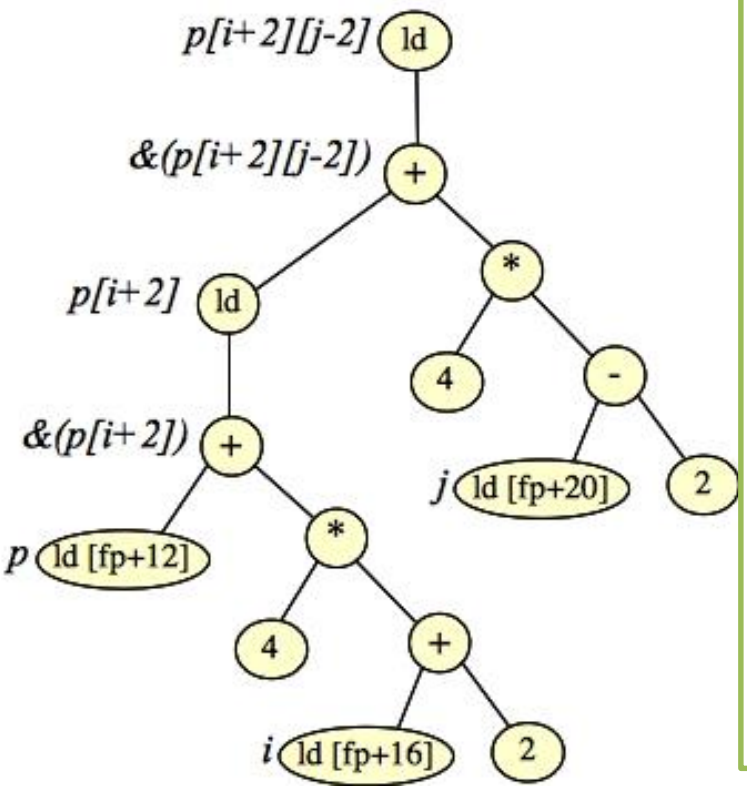
Double Indirection

```
main()
{
  int a[3], b[3], c[3];
  int *d[3]; int e;

  a[0] = 1; a[1] = 2; a[2] = 3;
  b[0] = 4; b[1] = 5; b[2] = 6;
  c[0] = 7; c[1] = 8; c[2] = 9;

  //d[3][3]
  d[0] = a; d[1] = b; d[2] = c;

  e = x(d, 0, 3);
}
```

```
main:
   push #52
   st %g1 -> [fp-48]     / Do a[0] through c[2].
   mov #2 -> %r0
   st %r0 -> [fp-44]
   mov #3 -> %r0
   st %r0 -> [fp-40]
   mov #4 -> %r0
   st %r0 -> [fp-36]
   mov #5 -> %r0
   st %r0 -> [fp-32]
   mov #6 -> %r0
   st %r0 -> [fp-28]
   mov #7 -> %r0
   st %r0 -> [fp-24]
   mov #8 -> %r0
   st %r0 -> [fp-20]
   mov #9 -> %r0
   st %r0 -> [fp-16]

   mov #-48 -> %r0          / d[0] = a
   add %fp, %r0 -> %r0
   st %r0 -> [fp-12]

   mov #-36 -> %r0          / d[1] = b
   add %fp, %r0 -> %r0
   st %r0 -> [fp-8]

   mov #-24 -> %r0          / d[2] = c
   add %fp, %r0 -> %r0
   st %r0 -> [fp-4]

   mov #3 -> %r0            / Push the arguments in reverse order
   st %r0 -> [sp]--
   st %g0 -> [sp]--
   mov #-12 -> %r0
   add %fp, %r0 -> %r0
   st %r0 -> [sp]--

   jsr x                       / call x and set e
   pop #12
   st %r0 -> [fp]
   ret
```

Double Indirection

```
int x(int **p, int i, int j)
{
  return p[i+2][j-2];
}
```




```
x:
   st %r2 -> [sp]--          / Spill r2

   ld [fp+20] -> %r0         / Do the right part of the tree.
   mov #2 -> %r1
   sub %r0, %r1 -> %r0
   mov #4 -> %r1
   mul %r0, %r1 -> %r0

   mov #2 -> %r1             / Do the left part of the tree
   ld [fp+16] -> %r2
   add %r1, %r2 -> %r1
   mov #4 -> %r2
   mul %r1, %r2 -> %r1
   ld [fp+12] -> %r2
   add %r1, %r2 -> %r1
   ld [r1] -> %r1

   add %r0, %r1 -> %r0       / Add them up
   ld [r0] -> %r0

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