Assembler 1

- Assembler simulator
- Registers
- Instructions
- Example code with global variables
- Example code with local variables

Assembler

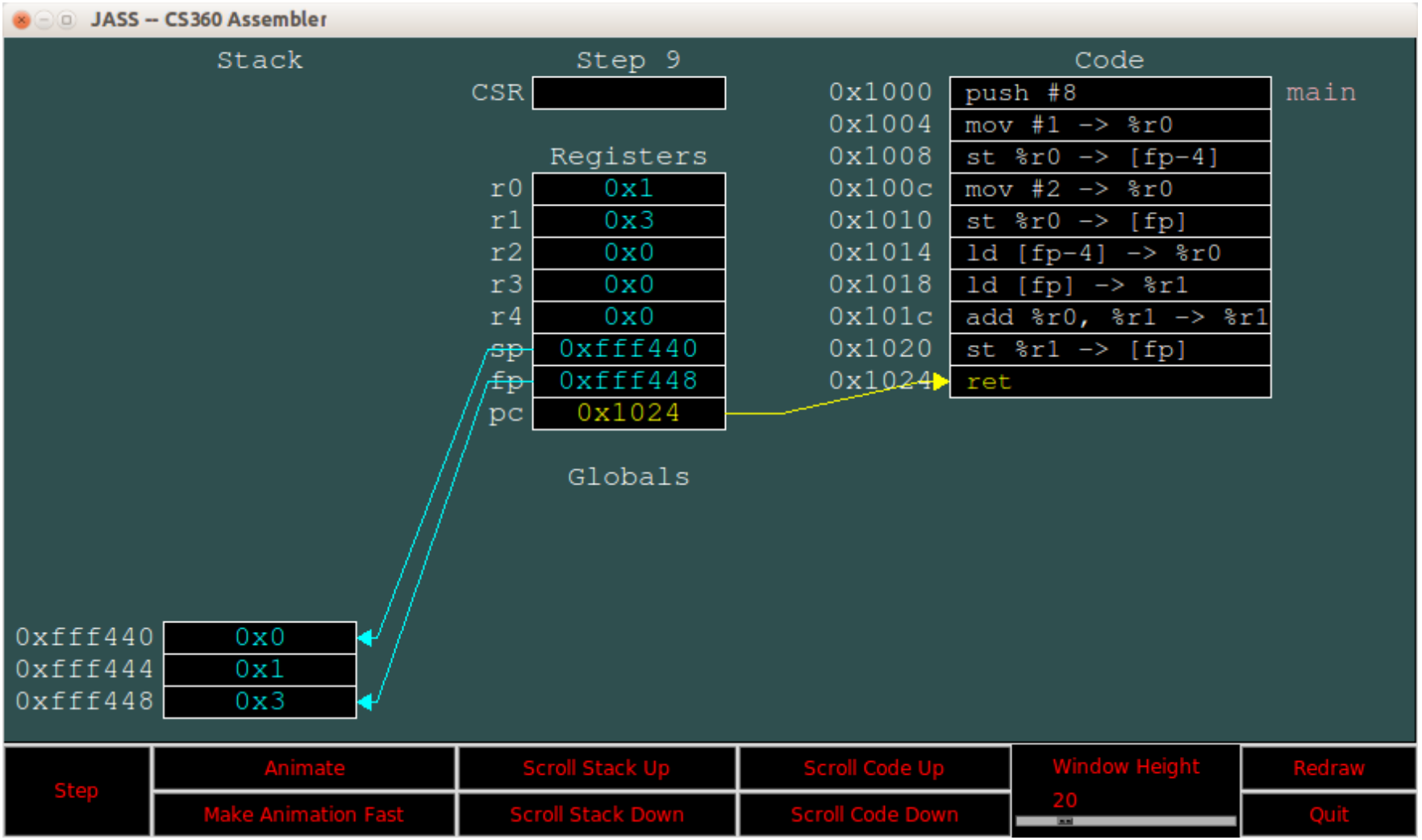
- The goal of this section is to teach you how assembly code is generated by a compiler to work on a standard processor.
- The assembly code is a made-up RISC assembly code that works on a fictitious machine that has 4-byte pointers and no floating point
- We will use a visual assembler for it

Registers

- We'll assume that our machine has 8 general-purpose registers in the CPU. All are 4 bytes, and can be read or written by the user. The first five are named **r0**, **r1**, **r2**, **r3**, **r4**. The last three registers are special:
- The sixth is named **sp** and is called the "*stack pointer*."
- The seventh is named **fp** and is called the "*frame pointer*."
- The eighth is named **pc** and is called the "*program counter*."

Assembler simulator

```
>>wish jassem.tcl
```



Registers

- Additionally, the computer has three read-only registers, which always contain the same values:
- **g0**, whose value is always zero.
- **g1**, whose value is always one.
- **gm1**, whose value is always negative one.
- Finally, the computer also has two special registers that the user cannot access directly:
- **IR** -- The instruction register. It holds the instruction currently being executed.
- **CSR** -- The control status register. It contains information pertaining to the execution of the current and previous instructions.

The instruction cycle

- The computer's operation consists of running instructions repetitively.
- This is known as the instruction cycle.
- The instruction cycle consists of 4 general phases:
- 1. Decode instruction (in IR)
- 2. Execute instruction
- 3. Determine next instruction
- 4. Load next instruction into the IR
- Like everything else, it's a sequence of 0's and 1's. We're going to assume that all of our instructions are 32 bits (although that's a naive assumption, it will work for our purposes).
- Instructions are stored as part of a program's memory, and the instruction that is pointed to by the pc register is the one that gets loaded into the IR for execution.
- In other words, if the pc contains the value 0x2040, then the IR is executing the instruction contained in the 4 bytes starting at memory address 0x2040.
- Assembly code is a readable encoding of instructions. A program called an assembler converts assembly code into the proper 0's and 1's that compose the program.
- If you call gcc with the -S flag, it will produce a .s file containing the assembler for that C program. Without the -S flag, it produces the instructions directly.

The instruction cycle

- Instructions are stored as part of a program's memory, and the instruction that is pointed to by the pc register is the one that gets loaded into the IR for execution.
- In other words, if the pc contains the value 0x2040, then the IR is executing the instruction contained in the 4 bytes starting at memory address 0x2040.
- Assembly code is a readable encoding of instructions. A program called an assembler converts assembly code into the proper 0's and 1's that compose the program.
- If you call gcc with the -S flag, it will produce a .s file containing the assembler for that C program. Without the -S flag, it produces the instructions directly.

Instructions: 1. Memory <-> Register instructions:

İşlem	Açıklama
ld mem -> %reg	Load the value of the register from memory.
st %reg -> mem	Store the value of the register into memory.
st %r0 -> i	Store the value of register r0 into the memory location of global variable i.
st %r0 -> [r1]	Treat the value of register r1 as a pointer to a memory location, and store the value of r0 in that memory location.
st %r0 -> [fp+4]	Treat the value of the frame pointer as a pointer to a memory location, and store the value of r0 in the memory location 4 bytes after that location. You can use any value, positive or negative – the value must be a multiple of four. However, you cannot use a register (i.e. you can't do st %r0 -> [fp+r2]). This only works with the frame pointer. It does not work with any other register.
st %r0 -> [sp]--	Treat the value of register sp as a pointer to a memory location, store the value of r0 into that memory location, and then subtract 4 to the value of sp.
st %r0 -> ++[sp]	Treat the value of register sp as a pointer to a memory location. First, add 4 to that value, then store the value of r0 into that memory location

Instructions: 2. Register <-> Register instructions:

İşlem	Açıklama
mov %reg -> %reg	Copy a register's value to another
mov #val -> %reg	register, or set its value to a constant.
add %reg1, %reg2 -> %reg3	reg3 = reg1 + reg2
sub %reg1, %reg2 -> %reg3	reg3 = reg1 - reg2
mul %reg1, %reg2 -> %reg3	reg3 = reg1 * reg2
idiv %reg1, %reg2 -> %reg3	reg3 = reg1 / reg2
imod %reg1, %reg2 -> %reg3	reg3 = reg1 % reg2
push %reg **push** #val	This subtracts the value of %reg or #val from the stack pointer.
pop %reg **pop** #val	This adds the value of %reg or #val to the stack pointer.

Instructions: 3. Control instructions

İşlem	Açıklama
jsr a	Call the subroutine starting at instruction a.
ret	Return from a subroutine.
.globl i	Allocate 4 bytes in the globals segment for the variable i.

The address space

- Each program's view of its memory is called an "address space". Typically an address space is broken up into 4 parts: The code, globals, heap, and stack. The code holds nothing but instructions.
- The globals is where global variables are stored, and the heap is where malloc'd storage lives. The stack is for temporary storage, like local variables and arguments for procedures.
- Generally, a process treats memory like a huge array of bytes, however, the bytes are organized logically into units of 4 bytes each, as that is the size of registers. We assume that this memory is of size 0x80000000 (this is hexadecimal). The code starts at address 0x1000 (typically, at a higher address, but for simplicity, we'll say 0x1000 here). The globals follow the code, and the heap follows the globals. As a program executes, the heap and stack might grow and shrink, but the code and globals stay the same size. The stack grows from back to front, starting at address 0x80000000 (actually, starting at 0x7fffffff), and growing towards the heap. In between the heap and stack is unused memory.

Simple compiled code.

The C compiler takes C code, and translates it into instructions. What we're doing in this and the following lectures is seeing how this translation works. The assembler code produced by the translation consists of machine instructions and directives. The translation is very logical.

C kodu	Assembly kod

int i;
int j;
main()
{
i = 1;
j = 2;
j = i + j;
}
	.globl i
	.globl j
	main:
	mov #1 -> %r0 / i = 1
	st %r0 -> i
	mov #2 -> %r0 / j = 2
	st %r0 -> j
	ld i -> %r0 / j = i + j
	ld j -> %r1
	add %r0,%r1 -> %r1
	st %r1 -> j
	ret

Simple compiled code.

Assembly kod	Assembly kod (optimize edilmiş)
.globl i .globl j main: mov #1 -> %r0 / i = 1 st %r0 -> i mov #2 -> %r0 / j = 2 st %r0 -> j ld i -> %r0 / j = i + j ld j -> %r1 add %r0,%r1 -> %r1 st %r1 -> j ret	.globl i .globl j main: mov #1 -> %r0 mov #2 -> %r1 add %r0,%r1 -> %r1 st %r1 -> j st %r0 -> i ret

Simple compiled code.

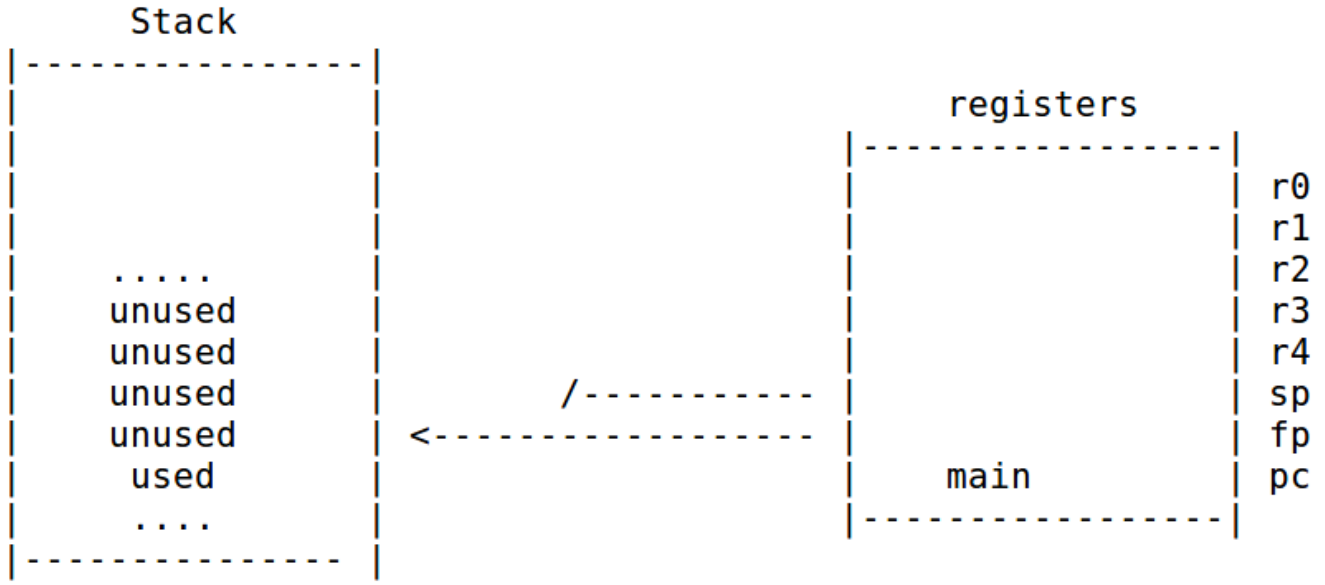
Now, the code for **main()** is just like before, only instead of accessing **i** and **j** as global variables, we access them as offsets to the frame pointer

C kodu	Assembly kodu

main()
{
int i, j;
 	
i = 1;
j = 2;
j = i + j;
}
	main:
	push #8 / This allocates i and j
	mov #1 -> %r0
	st %r0 -> [fp-4] / Set i to 1
	mov #2 -> %r0
	st %r0 -> [fp] / Set j to 2
	ld [fp-4] -> %r0
	ld [fp] -> %r1
	add %r0,%r1 -> %r1 / Add i and j and put the result
	st %r1 -> [fp] / back into j
	ret

Step by step execution

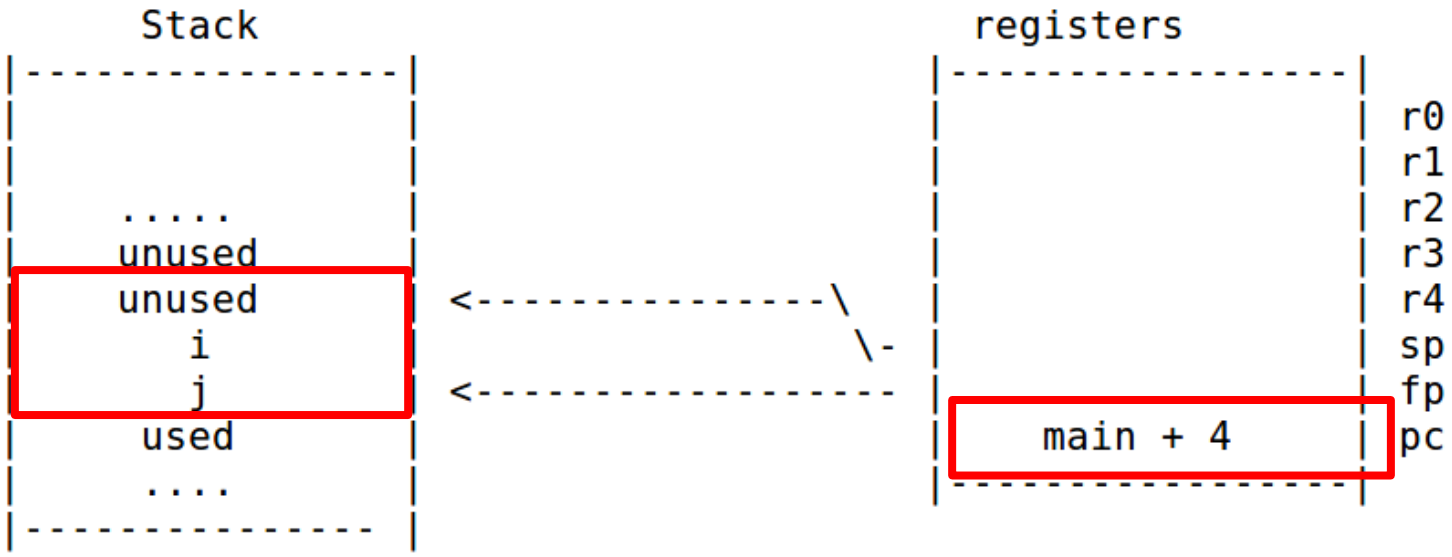
```
main:
        push #8             / This allocates i and j
        mov #1   -> %r0
        st  %r0 -> [fp-4]       / Set i to 1
        mov #2  -> %r0
        st  %r0 -> [fp]         / Set j to 2
        ld  [fp-4]  -> %r0
        ld  [fp]    -> %r1
        add %r0,%r1 -> %r1      / Add i and j and put the result
        st  %r1 -> [fp]           / back into j
        ret
```



Step by step execution

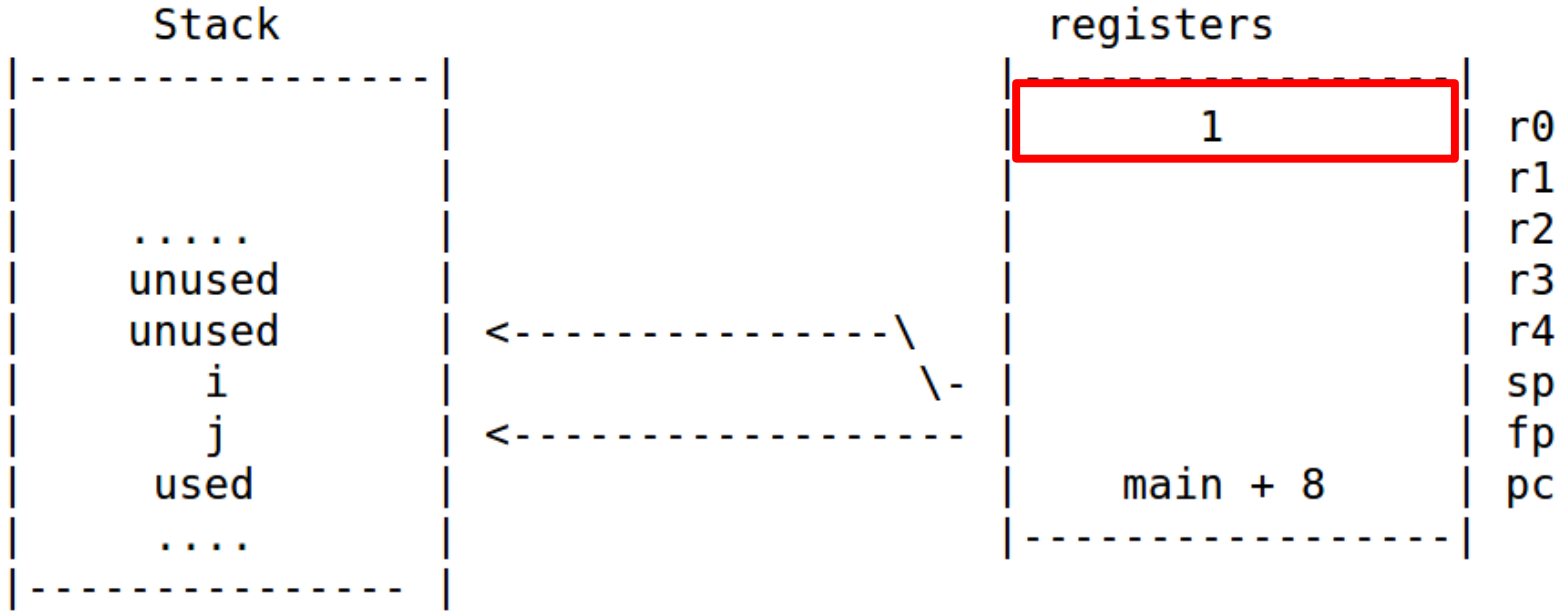
main:

```
push #8              / This allocates i and j
mov #1   -> %r0
st  %r0 -> [fp-4]      / Set i to 1
mov #2  -> %r0
st  %r0 -> [fp]        / Set j to 2
ld  [fp-4]  -> %r0
ld  [fp]    -> %r1
add %r0,%r1 -> %r1     / Add i and j and put the result
st  %r1 -> [fp]          / back into j
ret
```



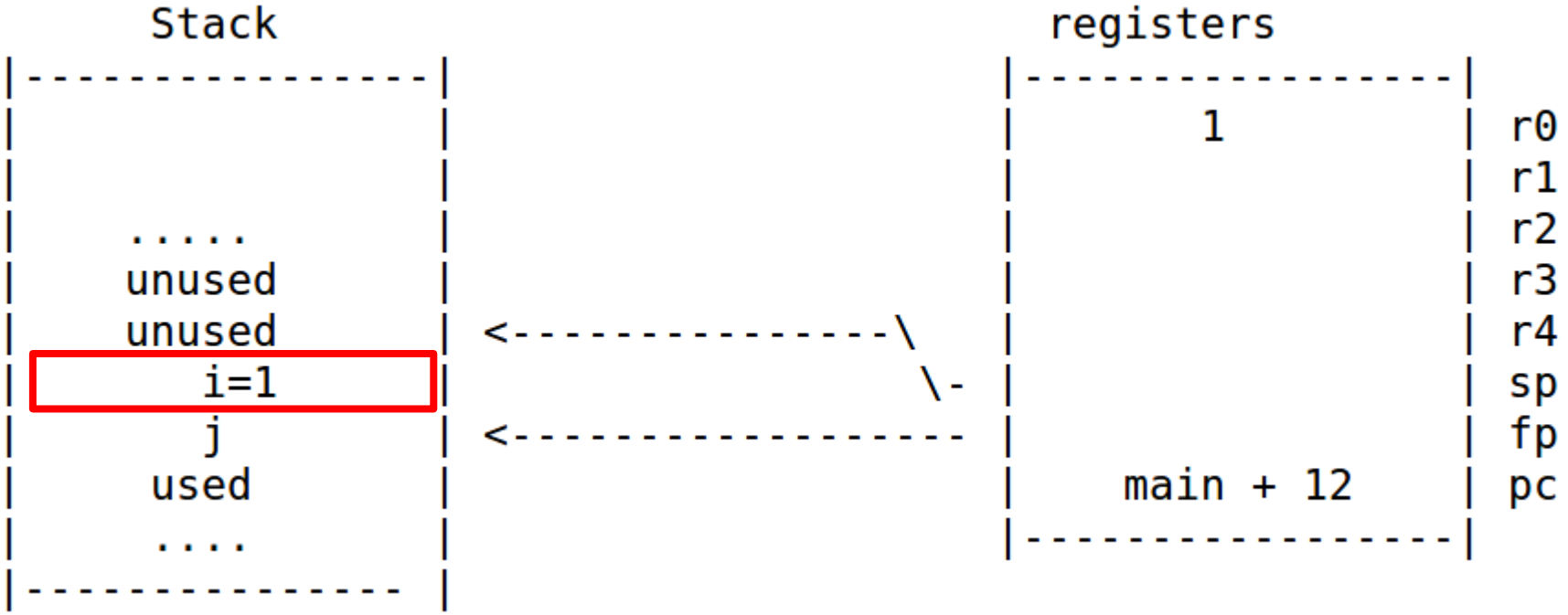
Step by step execution

```
main:
        push #8              / This allocates i and j
        mov #1   -> %r0
        st  %r0 -> [fp-4]      / Set i to 1
        mov #2  -> %r0
        st  %r0 -> [fp]        / Set j to 2
        ld  [fp-4]  -> %r0
        ld  [fp]    -> %r1
        add %r0,%r1 -> %r1     / Add i and j and put the result
        st  %r1 -> [fp]          / back into j
        ret
```



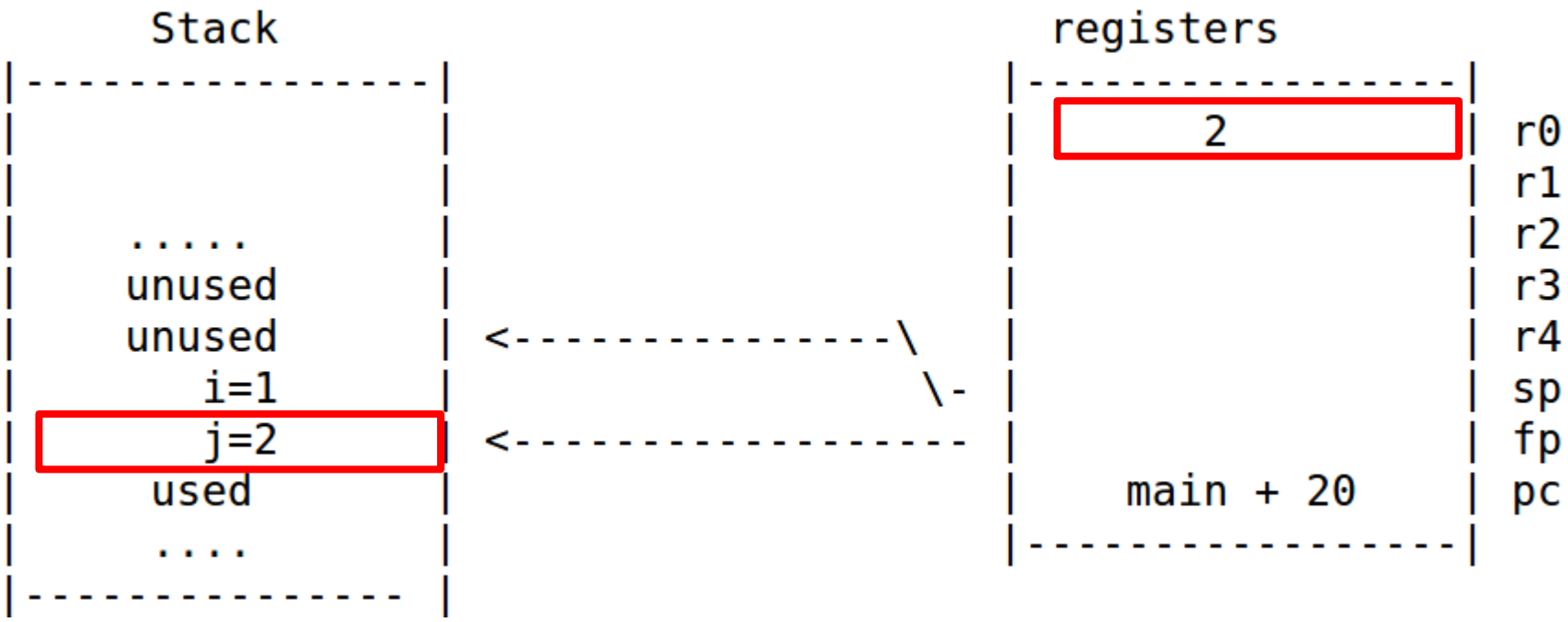
Step by step execution

```
main:
        push #8              / This allocates i and j
        mov #1   -> %r0
        st  %r0 -> [fp-4]      / Set i to 1
        mov #2  -> %r0
        st  %r0 -> [fp]        / Set j to 2
        ld  [fp-4]  -> %r0
        ld  [fp]    -> %r1
        add %r0,%r1 -> %r1     / Add i and j and put the result
        st  %r1 -> [fp]          / back into j
        ret
```



Step by step execution

```
mov 2    -> %r0
st  %r0 -> [fp]
```



Step by step execution

```
main:
        push #8             / This allocates i and j
        mov #1   -> %r0
        st  %r0 -> [fp-4]      / Set i to 1
        mov #2  -> %r0
        st  %r0 -> [fp]        / Set j to 2
        ld  [fp-4]  -> %r0
        ld  [fp]    -> %r1
        add %r0,%r1 -> %r1  / Add i and j and put the result
        st  %r1 -> [fp]           / back into j
        ret
```