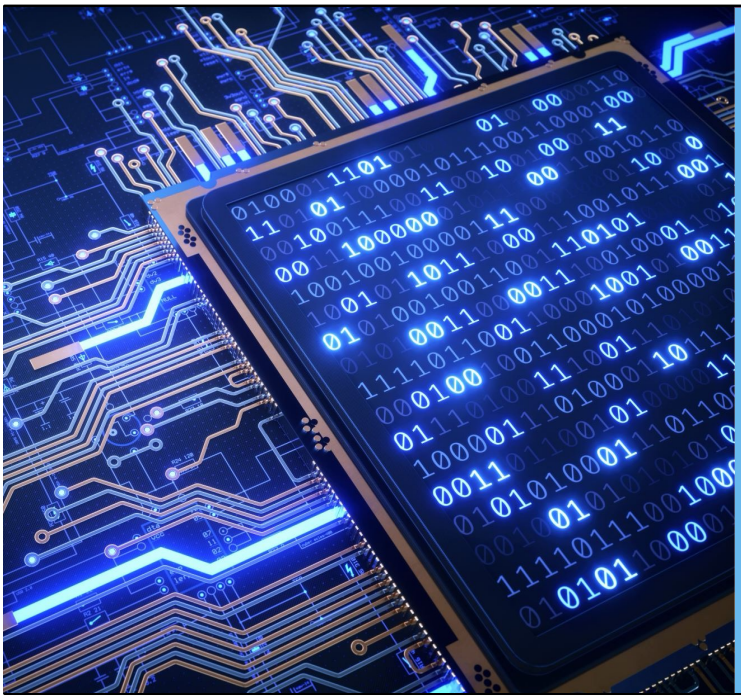


Computer Systems

CS51 Machine

CS51 – Fall 2026

Today we will talk about how CPUs run programs they read off memory.

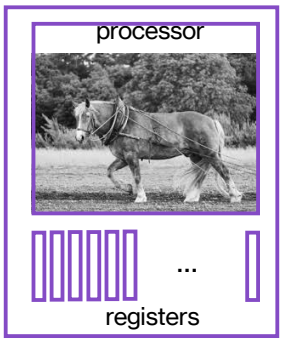
How does a program run on the CPU?

```
def add(num1, num2):
        return num1 + num2

def double(num):
        return 2 * num

def add_then_double(a, b):
        added = add(a, b)
        return double(added)

>> add_then_double(-10, 4)
```



How do programs run/execute on a computer?

We have experience with writing programs in programming languages like Python. How does our computer use the CPU to run/execute these programs?

Assembly code

- Python is a “high-level” programming language
- high-level programming languages allow you to write code:
  - without worrying about hardware-specific details of the computer (memory, registers, CPU specifics…)
  - higher-level abstraction, e.g., 2**6 or print()
- What actually runs on the processor is **assembly** code

Python is a high-level programming language which abstracts a lot of details and allows us to write programs that are less error-prone, without having to worry about hardware or all the details of what truly happens when we do fancy calculations or try to print messages. But in practice, the processor runs assembly code.

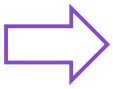
Assembly code

```
def add(num1, num2):
       return num1 + num2

def double(num):
       return 2 * num

def add_then_double(a, b):
       added = add(a, b)
       return double(added)

>> add_then_double(-10, 4)
```



```
add
    psh r2          ; save the return address
    loa r2 r1 4     ; get the second argument
    add r3 r3 r2    ; add r3 = arg1 + arg2
    pop r2          ; get the return address
    jmp r2          ; return

double
    psh r2          ; save the return address
    add r3 r3 r3    ; double the input2
    pop r2          ; get the return address
    jmp r2          ; return

add_then_double
    psh r2          ; save the return address
    loa r2 r1 4     ; get the second argument

    psh r2          ; setup call for add
    lcw r2 add      ; r3 has input one, push input2 on stack
    cal r2 r2       ; call add
    pop r0          ; remove input2 from stack

    lcw r2 double   ; result of add is in r3
    cal r2 r2       ; call double

    pop r2          ; get the return address
    jmp r2          ; return
```

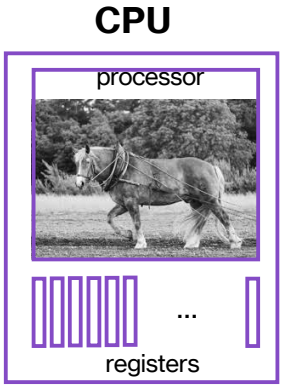
That means there is a conversion of code from languages like Python to an assembly language like the one on the right. As you can see assembly is still readable but definitely more complex.

This week

- Introduce the CS51 machine
- This is a simplified version of an assembly language
- It is a “simulator” that assumes a very simple CPU and memory setup

Our plan for this week is to get familiar with the CS51 machine which is a simplified version of an assembly language. In practice, it is a simulator that assumes a very simple CPU and memory setup and works very well (I think!) in introducing you to what happens at a low level.

Inside the CPU

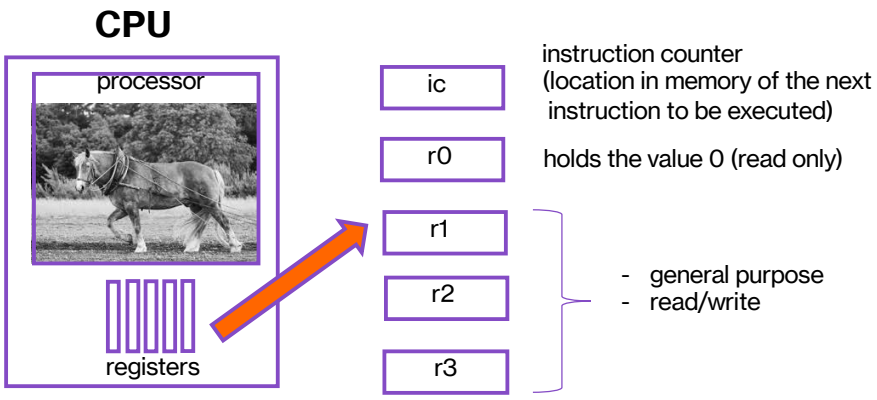


processor: does the work

registers: local, fast memory slots

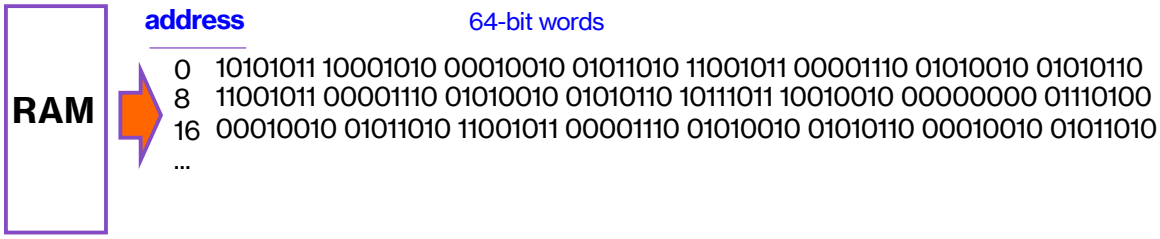
Going back to our CPU diagram, you can think of the processor as the workhorse that does all the work of executing programs. Registers on the other hand are very fast local memory slots that the processor utilizes to be able to execute programs.

CS51 machine (processor)



Specifically, for our own purposes, we will assume that there five registers. The first one, ic, stands for instruction counter and holds the location in memory of the next instruction in memory to be executed (it will make sense in a bit, I promise). The other four registers hold data values on which the machine operates. Register r0 is special in that it always contains the value zero; any attempt to change it is ignored. The other registers, r1, r2, and r3, are general-purpose registers whose values can be read and modified by program instructions.

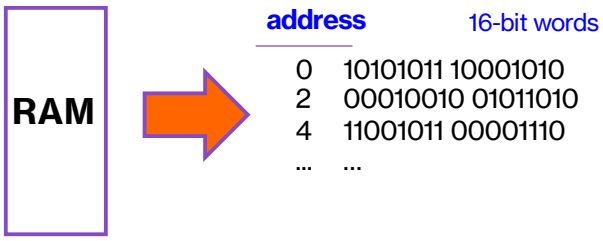
Memory



Most modern computers use 64-bit (8 byte) words

Remember last time we said that memory is organized in words and most modern computer architecture use 32-bit or more commonly these days 64-bit words.

Memory in the CS51 Machine

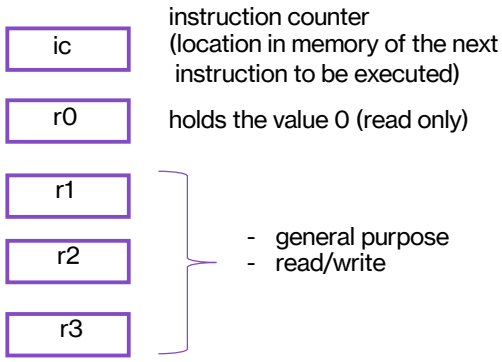


We'll use 16-bit (2-byte) words for our model (the CS51 machine)

To simplify things, our CS51 machine will assume 16-bit words, that is 2 bytes.

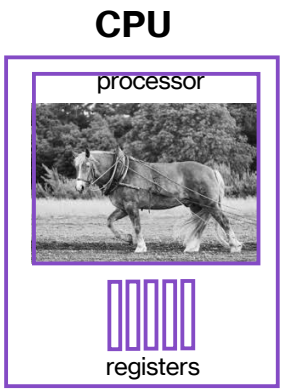
Program execution in the CS51 Machine

- When executing a program, the CS51 machine loops over the follow:
  - Fetch the value from `mem[ic]` for use as an instruction
  - Increment `ic` by 2
  - Decode the instruction and then execute it



When executing a program, the CS51 machine performs repeatedly the following steps:
1) First, it goes to the memory location indicated by the register ic and it fetches the value for use as an instruction to be executed. It then increments ic by 2 (why 2? because the word size is 2 bytes!). It proceeds with decoding and executing the instruction. And then moves to the next cycle of doing the same.

Think



What types of operations might we want to do (think really basic)?

Let's think. You are already familiar with writing programs in languages like Python. What types of operations might we want to support in our CS51 machine if we were to think of really basic functionality in code?

CS51 machine instructions

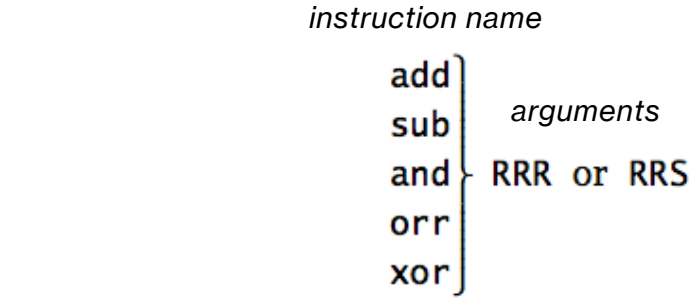
Four main types of instructions

1. math/logical
2. branch (conditionals, loops)
3. memory
4. control the machine (e.g., stop it)

In general, CS51 machine will support four types of instructions.

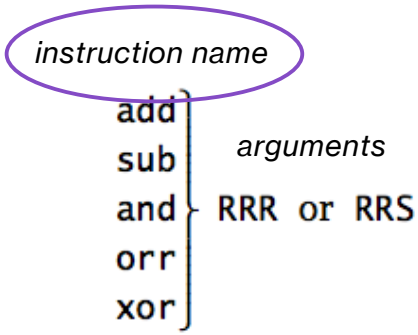
1. Math or logical operations (think additions, logical comparisons etc)
2. Branching in the form of control flow like conditionals and loops
3. Memory support (e.g., saving to or reading from memory)
4. and controlling the machine, e.g., to tell it to stop execution

Arithmetic and bitwise logical operations



Let's start with the math and bitwise logical operations

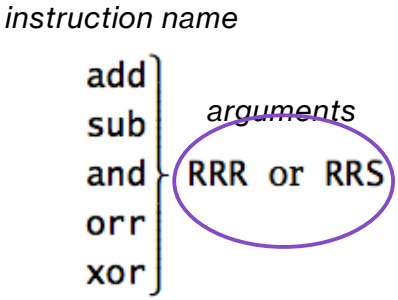
Arithmetic and bitwise logical operations



instruction/operation name is always three characters

We will soon talk about what each of these instructions do but for now I want you to notice that the instruction name is always three characters.

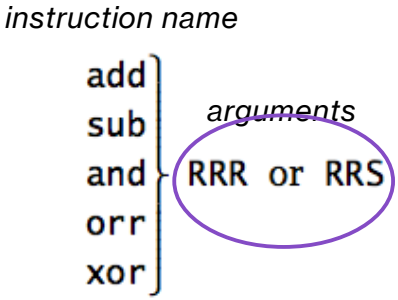
Arithmetic and bitwise logical operations



R = register (e.g. r0)
S = signed number (byte)

Next, let's look at the operation arguments which again are three characters. R stands for which register and S is a potential signed number.

Arithmetic and bitwise logical operations



1st R:	register where the answer will go
2nd R:	register of first operand
3rd S/R:	register/value of second operand

The first register corresponds to the register that the answer will go to. The second register is the register of the first operand. You can think of operand as the input to operator (think, parameters for functions). And finally the third option is either the register of the value of the second operand.

Think

```
add r1 r2 r3
```

What does this do?

1st R:	register where the answer will go
2nd R:	register of first operand
3rd S/R:	register/value of second operand

OK this was all rather abstract. Let's see an example of an instruction. What do you think that add r1 r2 r3 do?

Arithmetic and bitwise logical operations

```
add r1 r2 r3
```

r1 = r2 + r3

Add contents of registers r2 and r3 and store the result in r1

1st R:	register where the answer will go
2nd R:	register of first operand
3rd S/R:	register/value of second operand

It adds contents of register r2 and r3 and stores the result into r1

Think

```
add r2 r1 10
```

What does this do?

1st R:	register where the answer will go
2nd R:	register of first operand
3rd S/R:	register/value of second operand

ok, how about add r2 r1 10?

Arithmetic and bitwise logical operations

```
add r2 r1 10
```

r2 = r1 + 10

Add 10 to the contents of register r1 and store in r2

1st R:	register where the answer will go
2nd R:	register of first operand
3rd S/R:	register/value of second operand

It adds 10 to the contents of r1 and stores it in r2

Think

Hint: r0 is always 0

```
add r1 r0 8
sub r2 r0 r1
sub r2 r1 r2
```

What number is in r2?

1st R:	register where the answer will go
2nd R:	register of first operand
3rd S/R:	register/value of second operand

How about this small program. What number do you think is in r2? As a reminder, r0 is always 0

Think

Hint: r0 is always 0

```
add r1 r0 8
sub r2 r0 r1
sub r2 r1 r2
```

r1	8

What number is in r2?

1st R: register where the answer will go
2nd R: register of first operand
3rd S/R: register/value of second operand

add r1 r0 8 is equivalent to setting r1 to 8. adding a number or register to r0 is a common trick to set another one to the same value.

Think

Hint: r0 is always 0

```
add r1 r0 8
sub r2 r0 r1
sub r2 r1 r2
```

r1	8
r2	-8

What number is in r2?

1st R: register where the answer will go
2nd R: register of first operand
3rd S/R: register/value of second operand

sub r2 r0 r1 is equivalent to r2=r0-r1=0-8=-8. r1 remains unchanged

Think

Hint: r0 is always 0

```
add r1 r0 8
sub r2 r0 r1
sub r2 r1 r2
```

r1	8
r2	16

What number is in r2?

16

1st R: register where the answer will go
2nd R: register of first operand
3rd S/R: register/value of second operand

sub r2 r1 r2 is equivalent to r2=r1-r2=8-(-8)=16. r1 remains unchanged

Think

Hint: r0 is always 0

```
add r1 r0 6
and r2 r1 10
add r3 r1 r2
```

What number is in r3?

1st R:	register where the answer will go
2nd R:	register of first operand
3rd S/R:	register/value of second operand

What about now, what's the number in r3 after this program executes?

Think

Hint: r0 is always 0

```
add r1 r0 6
and r2 r1 10
add r3 r1 r2
```

r1	6

What number is in r3?

1st R: register where the answer will go
2nd R: register of first operand
3rd S/R: register/value of second operand

add r1 r0 6 is equivalent to r1=r0+6=0+6=6

Think

Hint: r0 is always 0

```
add r1 r0 6
and r2 r1 10
add r3 r1 r2
```

r1	6

$6_{10} = 0110$
& $10_{10} = 1010$

What number is in r3?

1st R: register where the answer will go
2nd R: register of first operand
3rd S/R: register/value of second operand

and is our first logical operator. We will need to convert 6 and 10 to their binary representations. What is the result of 6&10?

Think

Hint: r0 is always 0

```
add r1 r0 6
and r2 r1 10
add r3 r1 r2
```

r1	6
r2	2

$6_{10} = 0110$
& $10_{10} = 1010$
$2_{10} = 0010$

What number is in r3?

1st R: register where the answer will go
2nd R: register of first operand
3rd S/R: register/value of second operand

it's 0010 or 2 in decimal which is stored in r2

Think

Hint: r0 is always 0

```
add r1 r0 6
and r2 r1 10
add r3 r1 r2
```

r3 = r1 + r2

r1	6
r2	2
r3	8

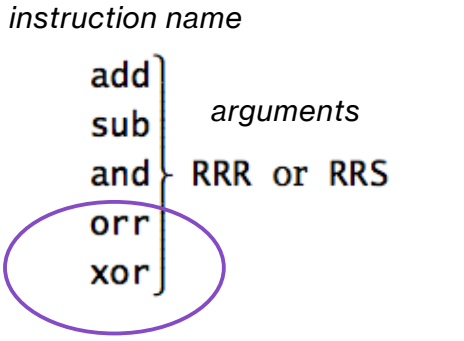
What number is in r3?

8

1st R: register where the answer will go
2nd R: register of first operand
3rd S/R: register/value of second operand

The last instruction to evaluate is add r3 r1 r2 which is equivalent to saying r3=r1+r2=6+2=8. So the final answer is that r3 will hold the number 8

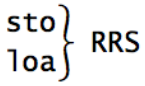
Arithmetic and bitwise logical operations



R = register (e.g. r0)
S = signed number (byte)

orr and xor work similarly to and. You need to convert the numbers to binary to know what the effect will be.

Memory and data-moving instructions

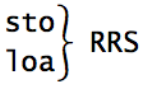


`sto` = save data in register TO memory
`loa` = put data FROM memory into a register

```
sto  r1  r2   ; store the contents of r1 to mem[r2]
loa  r1  r2   ; get data from mem[r2] and put into r1
```

Beyond math/logical operations, our CS51 machine supports operations that help us access the memory. Specifically, we have two instructions, sto and loa. sto r1 r2 stores the contents of r1 to the memory location stored in r2. loa r1 r2 retrives the data from mem[r2] and puts them into r1.

Special cases for memory access



`sto` = save data in register TO memory
`loa` = put data FROM memory into a register

- saving TO (sto) address 0 (r0) prints, e.g., `sto r1 r0`
- loading from (loa) address 0 (r0) gets input from user, e.g., `loa r1 r0` reads number from user and stores it in r1

The are two special cases when we use sto and loa and r0. If we have an instruction that the second register to store to is r0 then instead we print, e.g., sto r1 r0 results in the contents of register r1 being printed for the user to see. The second exception has to do when reading from r0. This prompts the user to enter their input. For example, loa r1 r0 prompts the user to give a number as an input and then that number is stored in register r1.

Basic structure of CS51 program

```
; great comments at the top!
;
        instruction1            ; comment
        instruction2            ; comment
        ...
label1
        instruction3            ; comment
        instruction4            ; comment
label2
        ...
        hlt
```



- whitespace before operations/instructions
- labels go here

This is what the basic structure of a CS51 machine looks like. Comments are signified with semicolon (;) and ignored. Instructions need to have whitespace before them. The instruction hlt terminates the program. You may also want to label chunks of code. For example, label1 might be useful if you want to start executing the code at instruction3

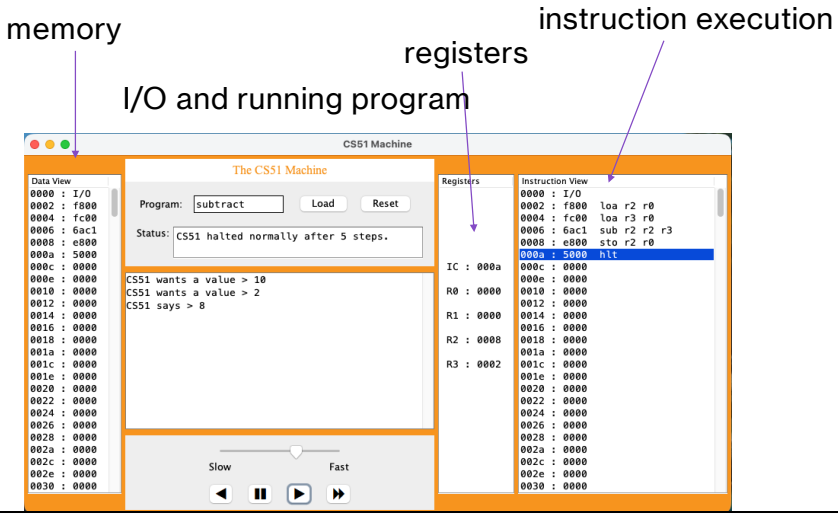
subtract.a51

```
; A simple CS51 Machine program that subtracts
; two numbers.

        loa r2 r0           ; get first value
        loa r3 r0           ; get second value
        sub r2 r2 r3        ; subtract them
        sto r2 r0           ; print result
        hlt                 ; quit
```

Let's look at our first example of a CS51 machine program. CS51 machine programs have the special extension .a51 (similar to how Python programs have .py etc) You see that at the top of our file we have two commented lines that explain the purpose of the program. The program itself only contains five instructions. loa r2 r0 prompts the user for a number which is stored in r2. load r3 r0 prompts the user for a second time and the second value is stored in register r2. sub r2 r2 r3 is equivalent to saying r2=r2-r3. This is the meat of the work we will do, subtracting the second input from the first. sto r2 r0 will take the result of the subtraction which is stored in r2 and print it to the user. hlt will quit the program.

CS51 simulator



We can see how that plays out in the CS51 machine. The simulator is divided in a few panels. On the left, you can see the memory under "Data view". This is in hex. The central console shows what program we have loaded (subtract.a51), the status: in this case we have run all steps. You see that the user was prompted to enter two values and that CS51 printed back that the result of 10-2 is equal to 8. The most interesting panel is the right most where we get to see the contents of the five registers (ic, r—r3) and the instruction view. You can see. there that the memory advances 2 bytes at a time (because the word size of the CS51 machine is two bytes) and that bytes 0 and 1 are reserved for I/O (input and output). Our instructions start being loaded in memory from byte 2 and onward.

Control instructions

branch (always)	`brs`	`B`
branch if ==	`beq`	`RRB`
branch if !=	`bne`	
branch if <	`blt`	
branch if >=	`bge`	
branch if >	`bgt`	
branch if <=	`ble`	

1st R: first register for comparison
2nd R: second register in comparison
3rd B: label

The CS52 Machine contains these familiar short branch instructions. brs B takes no register arguments, it always brunches to the part of the program labeled B. The others take two registers whose values are compared to decide whether or not to take the branch.

Think

```
beq r3 r0 done
```

What does this do?

1st R:	first register for comparison
2nd R:	second register in comparison
3rd B:	label

What do you think beq r3 r0 done does?

Think

```
beq r3 r0 done
```

If r3 == 0, branch to the label “done”
if not (else) ic is incremented as normal to the next instruction

1st R:	first register for comparison
2nd R:	second register in comparison
3rd B:	label

beq r3 r0 equal checks whether r3==0, and if yes, it moves the execution of the program to the first instruction after the label "done". If r3!=0 then the register ic is incremented as normal by 2 bytes to the next instruction

Think

```
ble r2 r3 done
```

What does this do?

1st R:	first register for comparison
2nd R:	second register in comparison
3rd B:	label

How about ble r2 r3 done

Think

```
ble r2 r3 done
```

If r2 <= r3, branch to the label “done”
if not (else) ic is incremented as normal to the next instruction

1st R:	first register for comparison
2nd R:	second register in comparison
3rd B:	label

It compares the contents of registers r2 and r3 and if r2 is less or equal to r3, then it branches to the label done. If not, it continues to the next instruction by advancing the ic by 2 bytes.

Control instructions

branch (always)	`brs`	`B`
branch if ==	`beq`	`RRB`
branch if !=	`bne`	
branch if <	`blt`	
branch if >=	`bge`	
branch if >	`bgt`	
branch if <=	`ble`	

- conditionals
- loops
- change the order that instructions are executed

Control instructions allow us to control the flow of a program, for example by supporting conditionals, loops, or changing the order that instructions are executed.

More CS51 machine examples

- Look at max_simple.a51
  - Gets two values from the user
  - Compares them
  - Uses a branch to distinguish between the two cases
  - Goal is to get largest value in r3
  - print largest value

Let's look at the max_simple.a51 program together.

simple_max.a51

```
;
; simple program to compute the max of
; two numbers
;
        loa r2 r0       ; get the first value and put it in r2
        loa r3 r0       ; get the second value and put it in r3

        bge r3 r2 done  ; check if r3 >= r2, if so jump to done
        add r3 r2 0     ; r3 = r2, (r2 is larger so copy it)
done
        sto r3 r0
        hlt
```

Let's next look at simple_max.a51

loa r2 r0 Gets first value from the user

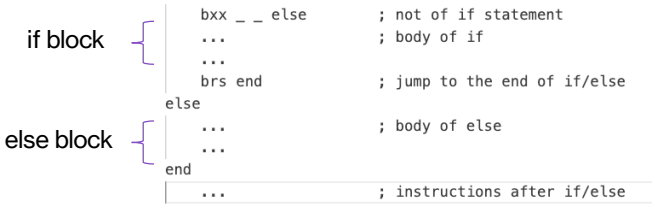
loa r3 r0 Gets the second value from the user

bge r3 r2 done Compares r3 and r2 and if r3>=r2 it moves the execution to label done, that is to the instruction sto r3 r0

If not r3>=r2 (that is r3<r2) it continues to the next instruction which sets r3 to r2 and the proceeds to the next

instruction sto r3 r0
So regardless of whether we branch or not, ultimately we will execute sto r3 r0 which means we will print r3 to the user and eventually terminate through hlt.
Putting it altogether, this program reads two numbers from the user and prints the larger one

if/else



- check the opposite of the if statement
  - if it is true, we'll jump down to else
  - if it is not true, we'll continue into the body of the if part
- At the end of the if block, need to jump to the end, otherwise, we'd continue onto else

control flow in cs51 machine can feel the exact opposite of what you're used to with if/else statements in other languages. We check the opposite of the if statement and if it's true we jump down to else instead of getting in the block. If not, then we'll continue into the body of the if part. At the end of the if block, we would need to jump to the end (otherwise we would execute all the code for else). we accomplish this by labeling the borders of the else block as end and we use brs end which always takes us there.

Think

```
    loa r3 r0

    and r2 r3 1
    beq r2 r0 else
    add r3 r0 47
    brs end
else
    add r3 r0 -47
end

    sto r3 r0
    hlt
```

What does this code do?

If that sounded confusing, let's try to see an example of a cs51 machine program. What do you think that this code do?

Think

```
    loa r3 r0

    and r2 r3 1
    beq r2 r0 else
    add r3 r0 47
    brs end
else
    add r3 r0 -47
end

    sto r3 r0
    hlt
```

if block

else block

Notice that we have an if and else block

if/else (even_commented.a51)

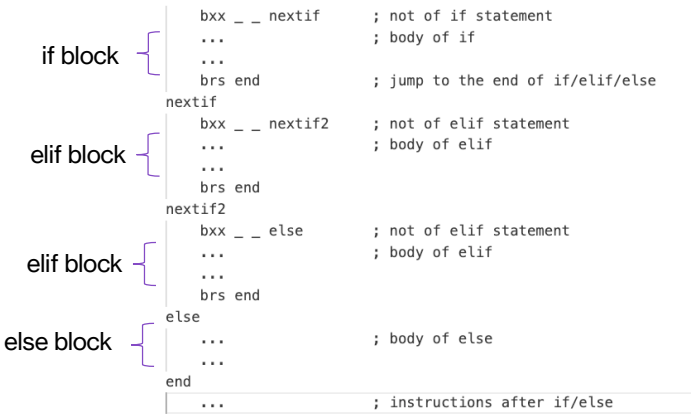
```
    loa r3 r0           ; get a value from the user

    and r2 r3 1         ; get the low-order bit into r2
    beq r2 r0 else      ; branch to else if even
    add r3 r0 47        ; put 47 in r3
    brs end             ; go to the end of the if/else
else
    add r3 r0 -47       ; put -47 in r3
end

    sto r3 r0           ; print out r3
    hlt
```

Let's walk through the entire program together (which you can find in the file even_commented.a51)
loa r3 r0 gets a value from the user and stores it in r3
and r2 r3 1 performs bitwise and with r3 and 1 and stores the result into r2. Hopefully it's clear that this bitwise operation isolates the least significant bit (right most bit, as this is the only bit that is coupled with 0..01)
beq r2 r0 else asks is r2==0? if yes, jump to else label, that is the add r3 r0 -47 that saves r3=-47 and we proceed with the next instruction sto r3 r0 which prints the contents of r3
if r2!=0 we continue to the next instruction add r3 r0 47 which sets r3=47. Next brs end jumps the execution to the label end that is sto r3 r0 which prints the contents of 3
Eventually we halt
So this programs prints 47 if the number is odd (last bit is 1) and -47 if it is even

If/elif/else



Beyond if/else control flow we can support if/elif/else using the same idea of labeling our code and carefully ensuring that we carefully exit the appropriate block so we don't end up executing all blocks following it (this is done through the brs end instruction)

Think

```
    loa r3 r0

    bge r3 r0 nextif
    add r3 r0 -1
    brs end
nextif
    bgt r3 r0 else
    add r3 r0 0
    brs end
else
    add r3 r0 1
end

    sto r3 r0
    hlt
```

What does this code do?

Let's see what that would look like in practice. What do you think the following code does?

if/elif/else (sign_commented.a51)

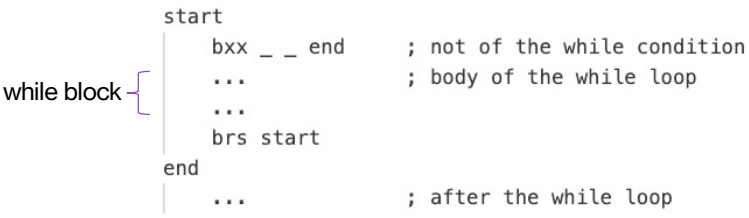
```
    loa r3 r0               ; get a number from the user

    bge r3 r0 nextif        ; if r3 < 0
    add r3 r0 -1            ; r3 = -1
    brs end
nextif
    bgt r3 r0 else          ; if r3 <= 0
    add r3 r0 0             ; r3 = 0
    brs end
else
    add r3 r0 1             ; r3 is positive: r3 = 1
end

    sto r3 r0               ; print out r3
    hlt
```

We will work through it together:
loa r3 r0 prompts the user for input and stores it in r3
bge r3 r0 nextif asks if r3>=0 and if yes, it jumps to next if
If not that is r3<0, it continues to add r3 r0 -1 which sets r3 to -1. It then jumps to end (brs end) which prints -1 and halts
Let's see what the nextif label does
We know that r3>=0. The first instruction in nextif is bgt r3 f0 else which asks whether r3>r0 and if yes, it jumps to else which is add r3 r0 1, or equivalently r3=1
if r3<=0, then it continues to add r3 r0 0 which is the same as setting r3=0 and then jumps to end which prints to use r3 (0) and halts
So the program prints either -1 or 0. When does it print -1? When the number the user gave us is negative! And 0 when it is positive

while loop



The last control flow we will support is a while loop. Again, we will play with labels to move the flow of our programs around. Notice that here for example, we have start and then at the end of the while block, brs start appears again, therefore looping us back to the first instruction of the start block. The only way to escape this loop is if our conditional bxx gets satisfied which then makes us jump to end and exits the loop.

Think

```
    loa r3 r0

    add r2 r0 0
start
    ble r3 r0 end
    add r2 r2 r3
    sub r3 r3 1
    brs start
end

    sto r2 r0
    hlt
```

What does this code do?

Your turn: what do you think this program do?

while loop (sum_commented.a51)

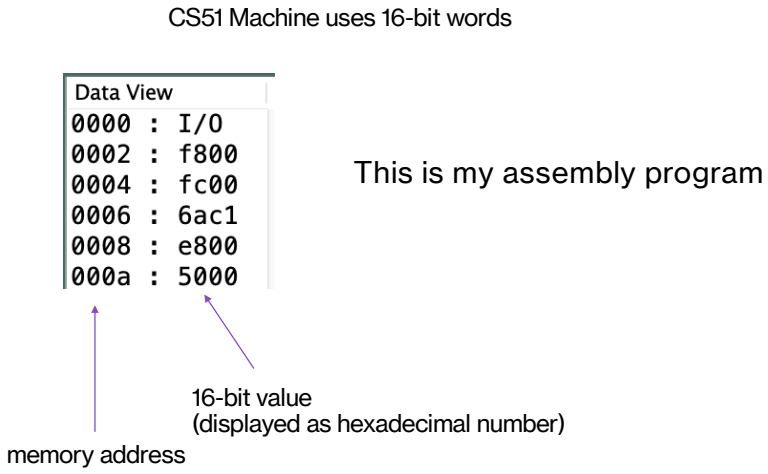
```
    loa r3 r0           ; get a number from the user

    add r2 r0 0         ; r2 = 0
start
    ble r3 r0 end       ; while r3 > 0
    add r2 r2 r3        ; r2 += r3
    sub r3 r3 1         ; r3 -= 1
    brs start
end

    sto r2 r0           ; print out r2
    hlt
```

Let's go over it together:
loa r3 r0 prompts the user for a number and stores it into r3
add r2 r0 0 sets r2=0
the start block signifies our while loop between start and end
ble r3 r0 end asks if r3<=0. If yes, it exist the loop and prints the number stored in r2
If not, that is r3>0 we keep going in the loop
add r2 r2 r3 is the same as r2=r2+r3
sub r3 r3 1 decrements r3 by 1, that is r3=r3-1
brs start brings us back to the beginning of the loop.
So overall what did we do? we took a number x from the user and we returned the sum 1+2+…+x to that number

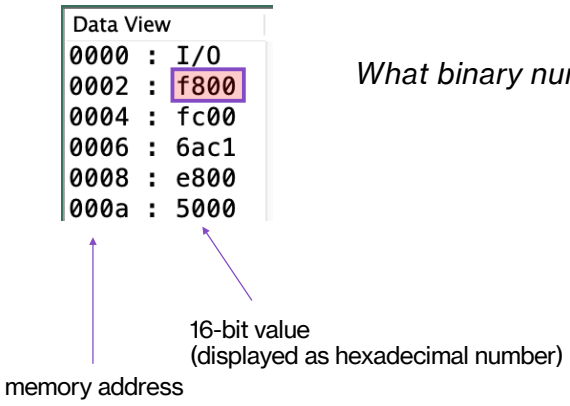
Instructions to binary



I skipped quickly what the simulator looks like on the left side. Let's revisit it. The data view shows the memory address (in hex). Since the word is 2 bytes long (16 bits), we see that we advance by 2.

Instructions to binary

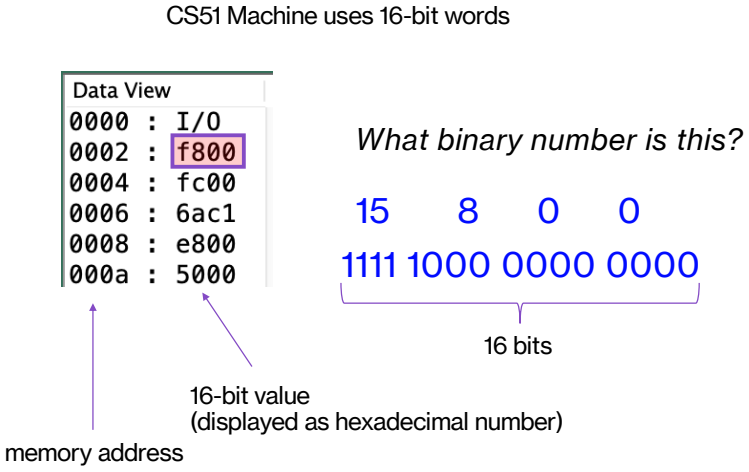
CS51 Machine uses 16-bit words



What binary number is this?

We already said that the first two bytes are reserved for I/O. Let's take the example of this program at address 2. The instruction is f800 in hex. What is its binary equivalent?

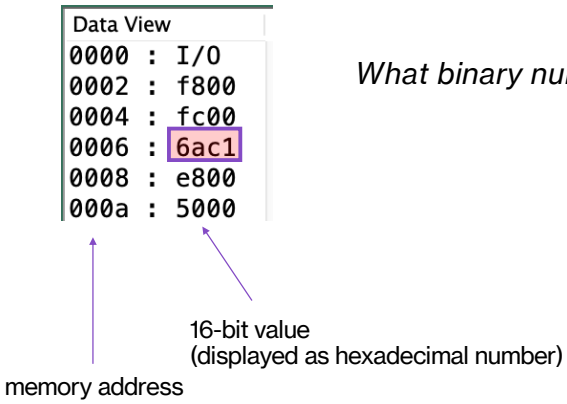
Instructions to binary



Remember, in hex it goes 0-9, a->10, b->11, c->12, d->13, e->14, f->15. So f is 15, 8 is 8, and the zeros are as is 9. If we convert each of these decimals to their unsigned representation, we have 1111 for 15, 1000 for 8, and 0000 for the 0

Instructions to binary

CS51 Machine uses 16-bit words

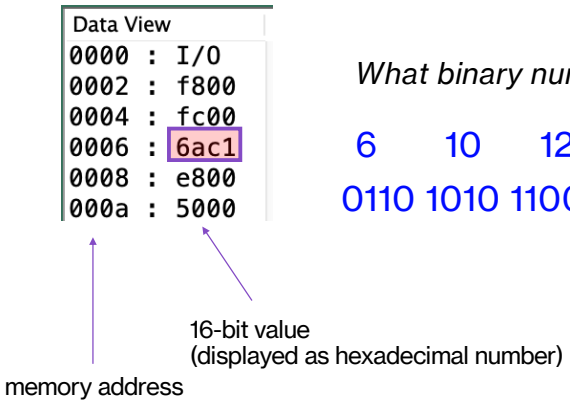


What binary number is this?

Let's do one more. What is 6ac1 in binary?

Instructions to binary

CS51 Machine uses 16-bit words



What binary number is this?

6 10 12 1

0110 1010 1100 0001

Following the same recipe, we translate a to 10 and c to 12 and then convert each of those decimal numbers to their binary representation.

Encoding instructions

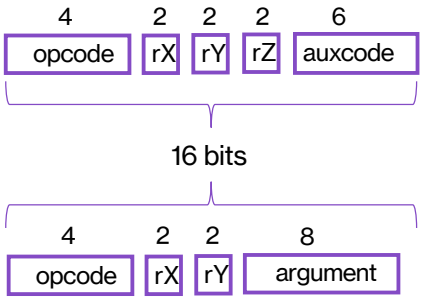
Two formats for instructions

opcode: specifies what operation (or category of operation)

r_: specifies a register

auxcode: specifies additional operations

argument: a number



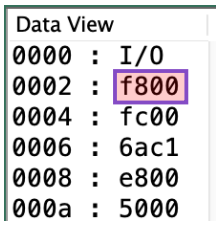
The CS51 machine follows two formats for its instructions. opcode specifies what operation (or category of operation) and it occupies 4 bits. the next 6 bits are distributed between 3 registers, each 2 bits long. the final part is a 6 bit auxcode that specifies the additional operations. Instead of this format, we could also have only two registers for 4 bits in total, followed by an 8-bit (1 byte) argument which corresponds to a number.

opcode

opcode	instruction
0x0	beq
0x1	bne
0x2	blt
0x3	bge
0x4	cal
0x5	hlt
0x6	arithmetic/logical instruction
...	
0xe	sto
0xf	loa

Here's what all combinations for opcodes and their matching instructions look like. For example 0000 corresponds to beq, 0001 to bne and so on

Instructions to binary



What binary number is this?

15 8 0 0

1111 1000 0000 0000

4 2 2 8

opcode | rX | rY | argument

Remember our example where we translated f8000 to its binary form?

Instructions to binary

Data View

```
0000 : I/O
0002 : f800
0004 : fc00
0006 : 6ac1
0008 : e800
000a : 5000
```

What is this instruction?

15 8 0 0

1111 1000 0000 0000

4	2	2	8
opcode	rX	rY	argument

loa r2 r0

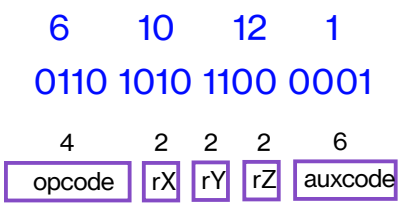
opcode	instruction
0x0	beq
0x1	bne
0x2	blt
0x3	bge
0x4	cal
0x5	hlt
0x6	arithmeti/logical instruction
...	
0xe	sto
0xf	loa

The first four bits correspond to loa, 10 is 2 in decimal that is register t2, 000 is 0 in decimal that is register 0

Instructions to binary

```
Data View
0000 : I/O
0002 : f800
0004 : fc00
0006 : 6ac1
0008 : e800
000a : 5000
```

What is this instruction?



opcode	instruction
0x0	beq
0x1	bne
0x2	blt
0x3	bge
0x4	cal
0x5	hlt
0x6	arithmetic/logical instruction
...	
0xe	sto
0xf	loa

How about this instruction?

Instructions to binary

```
Data View
0000 : I/O
0002 : f800
0004 : fc00
0006 : 6ac1
0008 : e800
000a : 5000
```

What is this instruction?

6 10 12 1

0110 1010 1100 0001

4	2	2	2	6
opcode	rX	rY	rZ	auxcode

arithmetic r2 r2 r3 0x1

opcode	instruction
0x0	beq
0x1	bne
0x2	blt
0x3	bge
0x4	cal
0x5	hlt
0x6	arithmetic/logical instruction
...	
0xe	sto
0xf	loa

0110 or 6 corresponds to an arithmetic instruction 10 is again r2 and 11 is 3 or r3. the last 6 bits corresponds to the auxcode 1

arithmetic/logical auxcode

auxcode	instruction
0x0	add
0x1	sub
0x2	
0x3	
0x4	and
0x5	orr
0x6	
0x7	
0x8	logical shift left
0x9	logical shift right
...	

Arithmetic and logical auxclodes include things like add, sub, and orr and logical shifts

Instructions to binary

```
Data View
0000 : I/O
0002 : f800
0004 : fc00
0006 : 6ac1
0008 : e800
000a : 5000
```

What is this instruction?

6 10 12 1

0110 1010 1100 0001

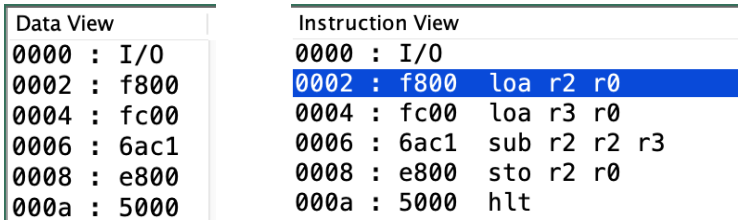
4	2	2	2	6
opcode	rX	rY	rZ	auxcode

sub r2 r2 r3

opcode	instruction
0x0	add
0x1	sub
0x2	
0x3	
0x4	and
0x5	orr
0x6	
0x7	
0x8	logical shift left
0x9	logical shift right
...	

So now we know that 0001 corresponds to sub and that whole thing makes 6ac1 equivalent to the instruction sub r2 r2 r3

instructions to binary



Indeed, you can see that the instructions correspond from the hex code in the data view to the instruction view on the right

Examples from this lecture

http://www.cs.pomona.edu/classes/cs51/cs51machine/

You can find all the examples we encountered today here along with a jar file that is the cs51machine program.