Recitation 11/6

Welcome back!

Directives

.text and .data directives - throwback to lecture 3

- Cannot run a program from just reading a .c file
- That’s what compilers (and other stuff) are for!
- Compiler: from .c file to assembly code
- Assembler: from assembly code to machine code (rep in binary) and load into memory according to the directives

Directives - cont'd

- Conceptually, memory is an array of bytes
- Directives help separate the different ways memory is used in a program
  - .text: code storage; memory you can only read or execute (run as a program)
  - .data: global and static variables; read-write memory, not executable
- Some memory segments are not specifically referenced by directives
  - Stack & Heap: read/write during execution, but also can be made not executable

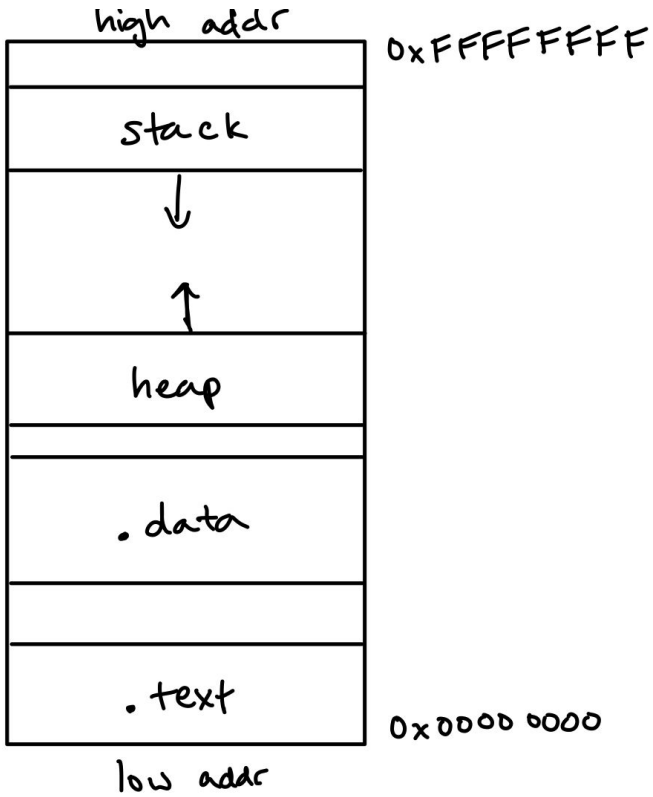


*diagram not to scale

Memory Separation

Security issue: what if a program accepts user input (typed into terminal), and malicious input is given?

Hacker: “here’s a string, but it’s actually code that if executed will cause your program to crash.”

Computer: “that’s ok, as long as I don’t move the instruction pointer to where your input is stored.”

Hacker: “Ok, then. What if I make the instruction pointer point to my code?”

.text permissions

Assumption #1: all code that should be executed was loaded into .text segment.

Therefore, .data segment, stack, and heap do not contain code that is intended to be executed by the running program

Why don’t we turn off execution permissions for those segments of memory?

Solution is known as Execution-Space Protection, or Windows calls it Data Execution Prevention (DEP)

.data, heap, and stack permissions

Assumption #2: the code, once loaded into .text, is not supposed to change.

Why don’t we turn off write permission for memory in the .text segment?

This is why .data and .text directives matter, by separating memory into different “functions,” you can specialize read-write-execute permissions and (to some extent) protect your program

Review of other directives

.globl : “global”

- Elevates status of symbol (function name or variable name) to global, so that you can use the symbol outside of this file
- I.e. if you want to use your swap function in your sort function but they’re in two separate files, it is possible to invoke swap if you first partially compile swap.s and and sort.s into swap.o and sort.o, and then link the two .o files to create sort executable. This is only possible if you made the swap function global, however.

.p2align: “power of 2 align”

- Given “.p2align x” in your code, every instruction will start on a memory address that is a multiple of 2^x
- We add “.p2align 2” to the beginning of our code because 1 instruction = 32 bits = 4 bytes = 2^2 bytes -> x = 2.
- Makes life easier for the hardware

Difference between directives and labels

Directives

- Finite set of commands for the assembler
- Not found in executable code
- Not all things that start with a period are directives!
- Both directives and labels are not indented, instructions underneath these are indented

Labels

- End with a colon (unless they're being referenced)
  - Do not do: `jal main:`
- Infinite number of possible labels because they are user-defined
- Save the address of certain instructions to make jumping and branching easier
  - Beginning of loops
  - Beginning of subroutines
  - Return to the caller function
  - if/else branches (conditionals)
- Function/subroutine labels are usually lowercase and do not start with a period (`main`, `swap`, `sort`)
- Other labels usually capitalized and start with a period (`.END`, `.LOOP`, `.ELSE`)

Connecting things together

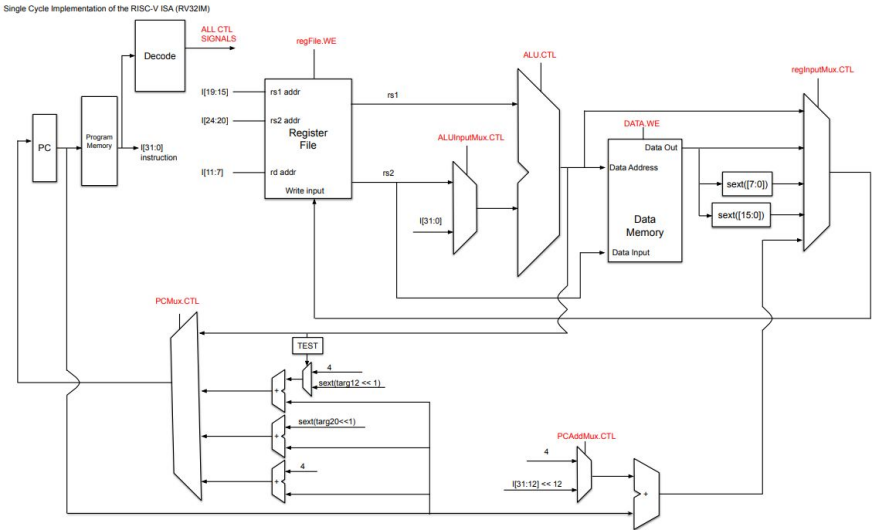
You may have done this for HW7

This is what HW8 is :)

Software → Assembly → Hardware

```
.globl main
.text
main:
        # Tests simple looping behaviour
        li t0, 60
        li t1, 0
loop:
        addi t1, t1, 5
        addi t0, t0, -1
        bne t1, t0, loop
        bne t1, zero, success
failure:
        li a0, 0
        li a7, 93
        ecall

success:
        li a0, 42
        li a7, 93
        ecall
```



Designing a processor

So given a list of assembly instructions, let’s design a processor that can perform operations using hardware components that are at our disposal.

Pumpkin soup

2 servings 15 minutes

INGREDIENTS

100 ml milk
50 g butter
3 eggs
1 tbs cocoa
2 tsp baking soda
a pinch of salt
3 eggs

NOTES

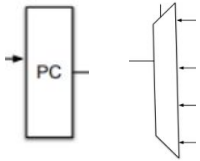
Nunc nulla velit, feugiat vitae ex quis, lobortis porta leo. Donec dictum lectus in ex accumsan sodales. Pellentesque habitant morbi tristique.

DIRECTIONS

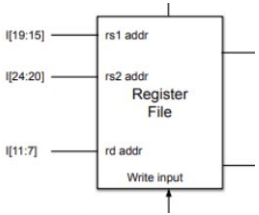
1. Nunc nulla velit, feugiat vitae ex quis, lobortis porta leo.
2. Donec dictum lectus in ex accumsan sodales. Pellentesque habitant morbi tristique.
3. Nunc nulla velit, feugiat vitae ex quis, lobortis porta leo. Donec dictum lectus in ex. lentesque habitant morbi tristique. Nunc nulla velit, feugiat vitae ex quis, lobortis porta leo. Donec dictum lectus in ex.
4. Habitant morbi tristique.Nunc nulla velit, feugiat vitae ex quis, lobortis porta leo. Donec dictum lectu,
5. Donec dictum lectus in ex accumsan sodales. Pellentesque habitant morbi tristique.
6. Nunc nulla velit, feugiat vitae ex quis, lobortis porta leo. Donec dictum lectus in ex. lobortis porta leo.

First, what do we want to achieve with the processor? And what hardware do we need for them?

Something to keep track of we are at, and where to go next

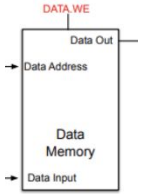


A temporary place for our operands



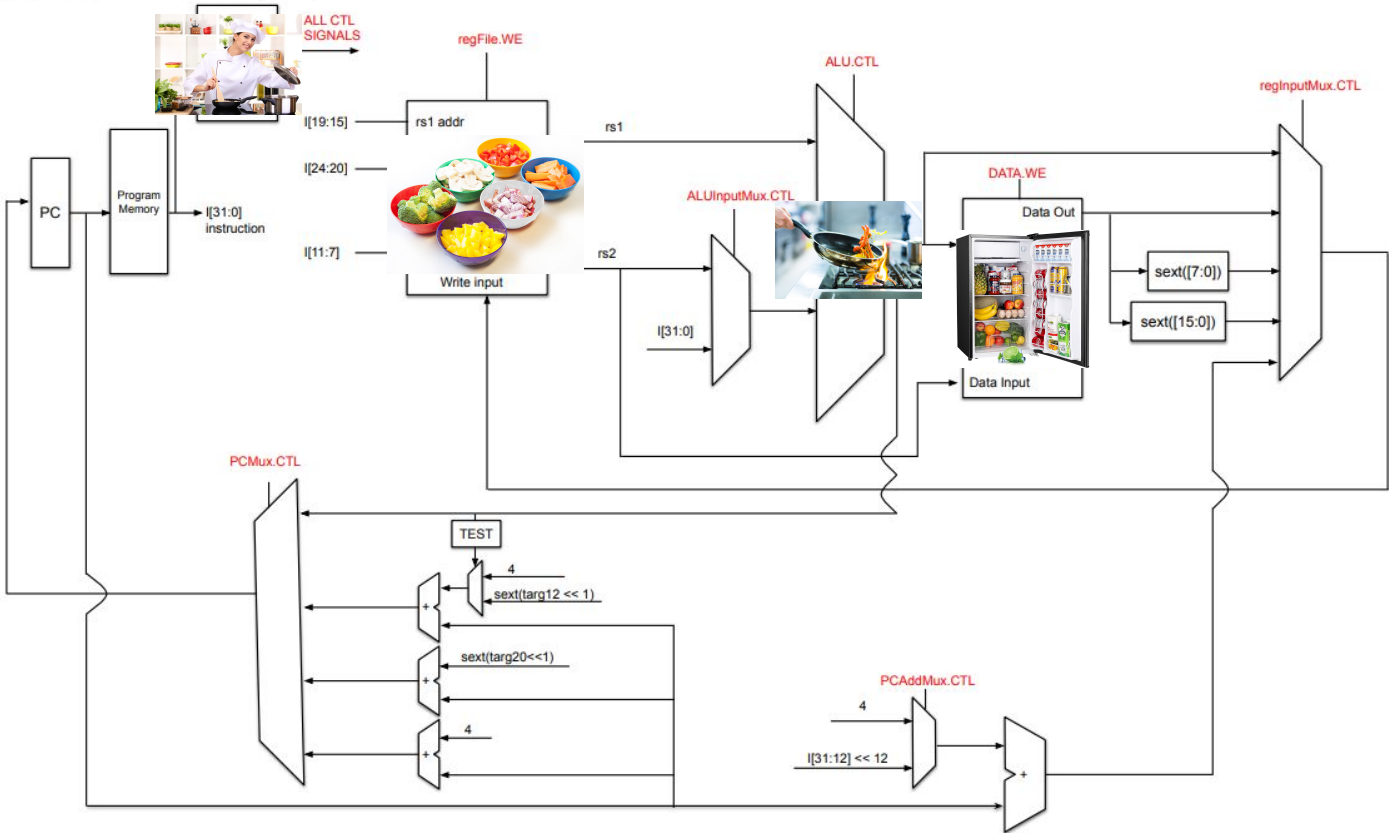
Something to perform the operation

A less temporary place to persist results



But how do we coordinate these components to perform the instruction?

Single Cycle Implementation of the RISC-V ISA (RV32IM)

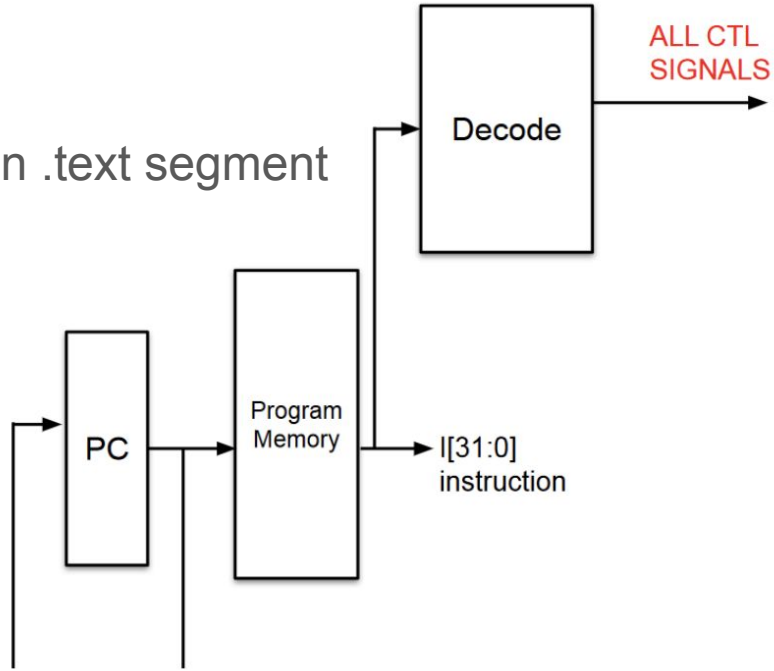


Decoder

PC
- pointer to the current instruction in .text segment

Program memory
- Byte-addressable
- 32-bit instructions

Decoder
- From instruction to signals
- this is your hw!



Why signals?

If you were the designer of the processor, and you want to control the processor, some questions you may wanna ask:

- Where do we get the inputs?
  - If inputs are hardcoded using immediate?
  - If inputs are from registers or from data memory?
- What operations do we perform?
- Do we save the result of the operation in memory?
- Where do we go after finishing the operation?

In other words, based on the 32-bit instruction, **how do we “give orders”** to the various parts of the processor?

What do the signals control?

- Registers
  - Do we write to the registers? - regFile.WE
  - What do we write back to the registers? - regInputMux.CTL
- Arithmetic Logic Unit
  - What are the inputs? - ALUInputMux.CTL
  - What operation do we perform on the inputs? - ALU.CTL
- Data memory
  - Do we update any data in the memory? - DATA.WE
- Branch Unit
  - How do we update PC? - PCMux.CTL

For a written explanation of how to determine control signal values, see EdPost #1303

Practice time ahhh

What are the control signals for each of these RISC-V instructions?

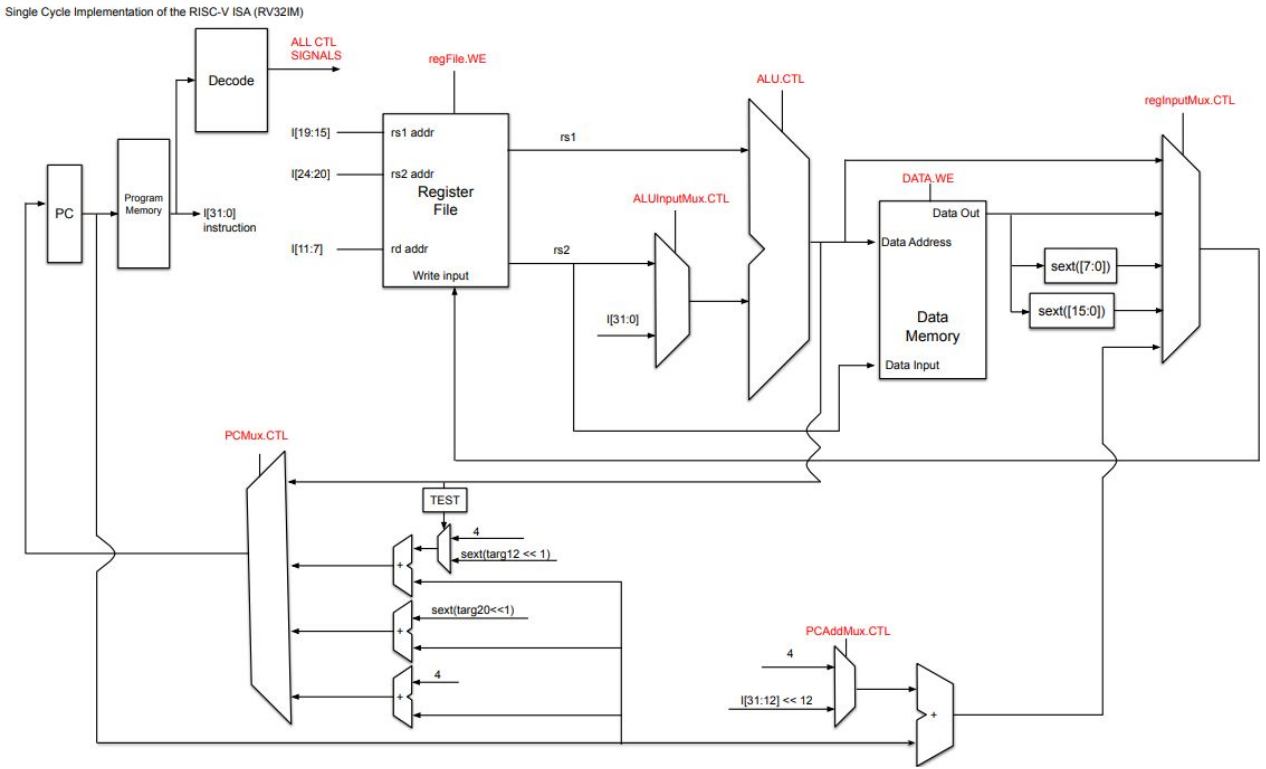
- auipc rd,imm20
- srl rd,rs1,rs2
- sb rs2,imm12(rs1)
- bge rs1,rs2,targ12

Example 1

instruction	fmt	opcode	fun3	fun7	semantics
auipc rd,imm20	U	17			rd = pc + (imm20 << 12)

31 25	24 20	19 15	14 12	11 7	6 0
imm[31:12]				rd	opcode	U-type

Control Signal	Value
regFile.WE	
regInputMux.CTL	
ALUInputMux.CTL	
ALU.CTL	
DATA.WE	
PCMux.CTL	
PCAddMux.CTL	

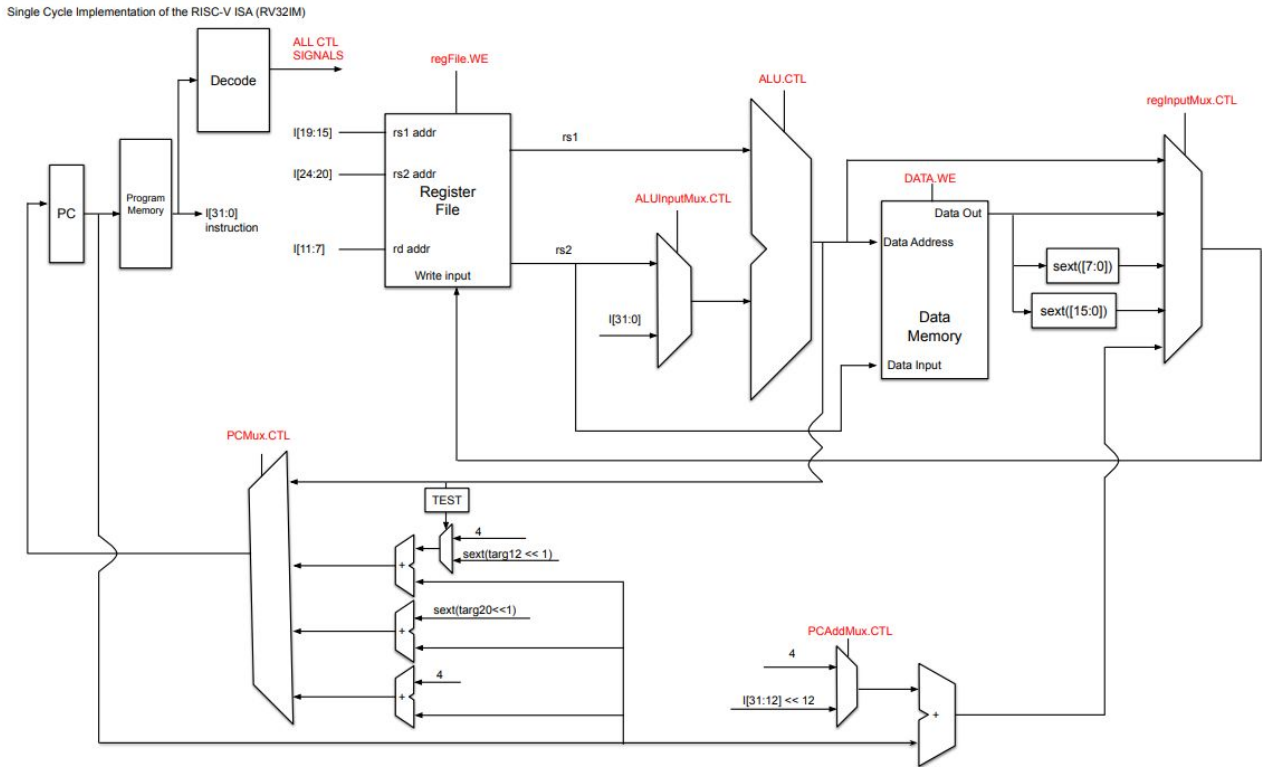


Sol 1

instruction	fmt	opcode	fun3	fun7	semantics
auipc rd,imm20	U	17			rd = pc + (imm20 << 12)

31 25	24 20	19 15	14 12	11 7	6 0
imm[31:12]				rd	opcode	U-type

Control Signal	Value
regFile.WE	1
regInputMux.CTL	4
ALUInputMux.CTL	X
ALU.CTL	X
DATA.WE	0
PCMux.CTL	3
PCAddMux.CTL	1

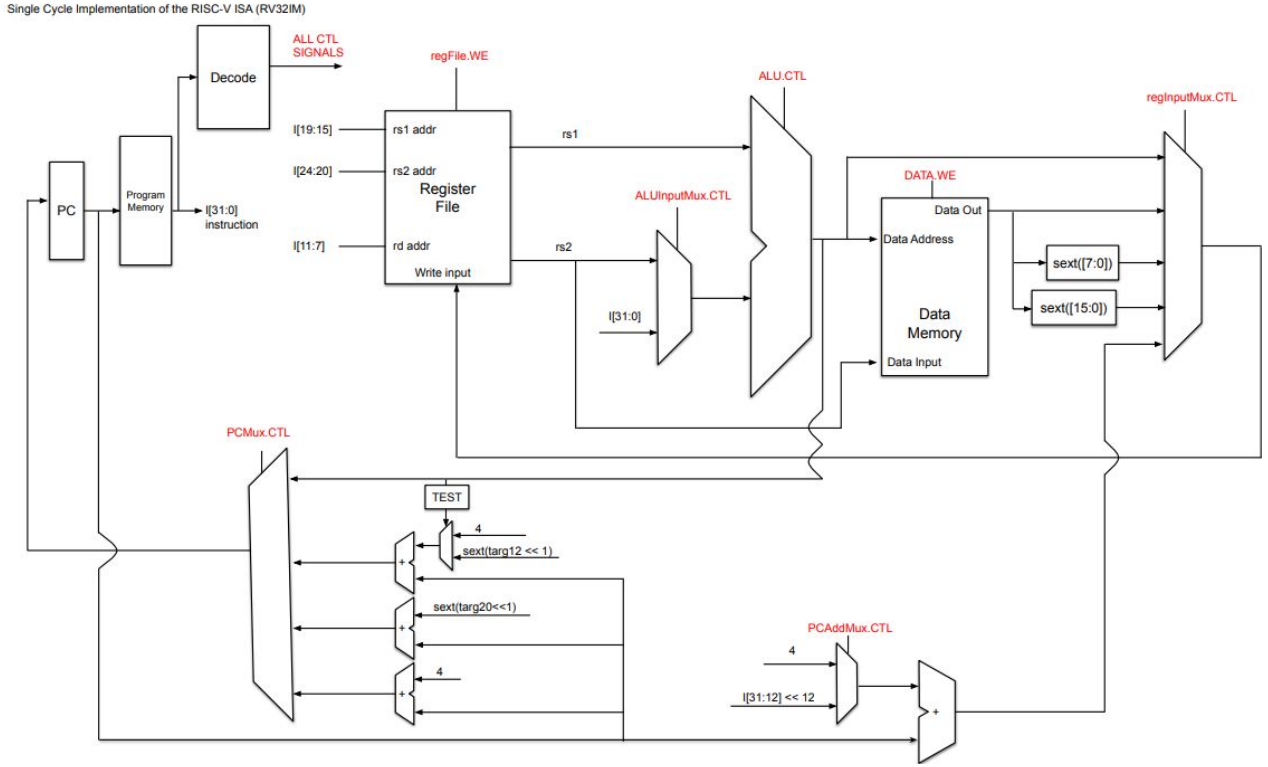


Example 2

Control Signal	Value
regFile.WE	
regInputMux.CTL	
ALUInputMux.CTL	
ALU.CTL	
DATA.WE	
PCMux.CTL	
PCAddMux.CTL	

instruction	fmt	opcode	fun3	fun7	semantics
srl rd,rs1,rs2	R	33	101	0x0	rd = rs1 >> rs2[4:0]

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode	R-type



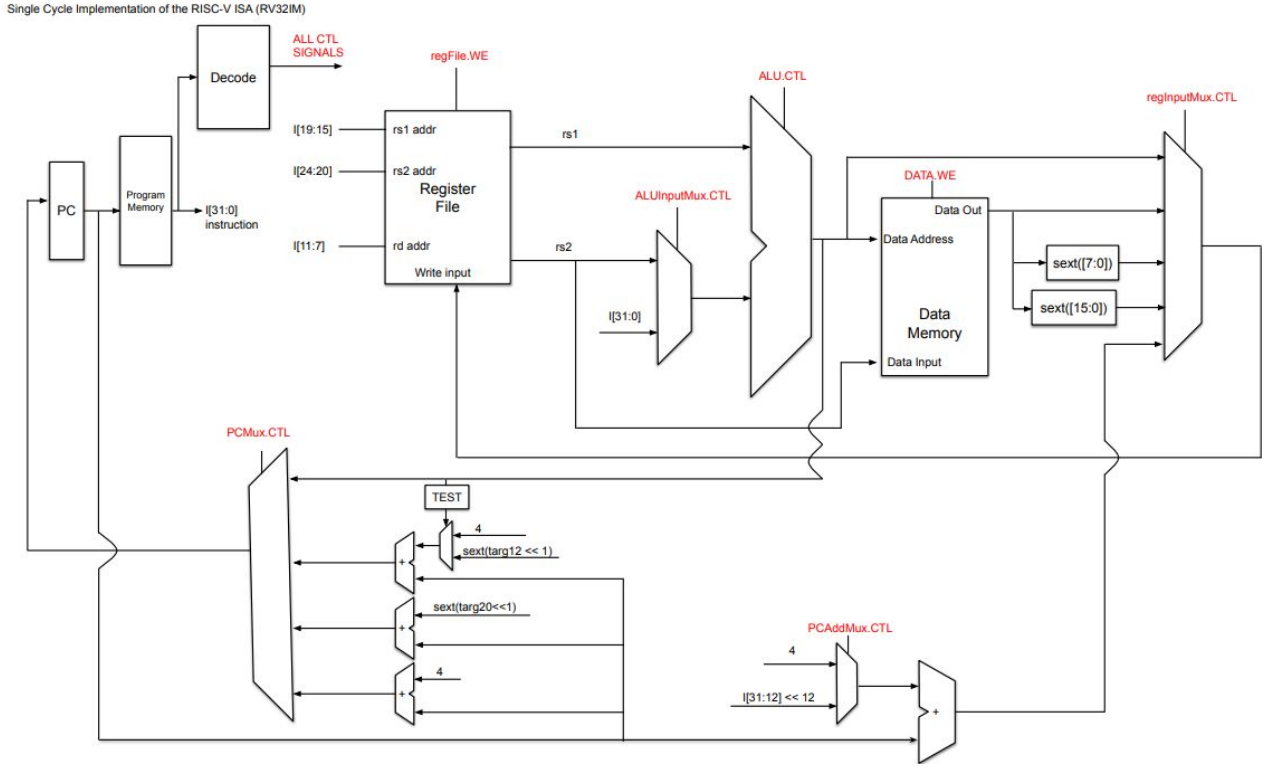
Sol 2

Control Signal	Value
regFile.WE	1
regInputMux.CTL	0
ALUInputMux.CTL	0
ALU.CTL	15
DATA.WE	0
PCMux.CTL	3
PCAddMux.CTL	X

instruction	fmt	opcode	fun3	fun7	semantics
srl rd,rs1,rs2	R	33	101	0x0	rd = rs1 >> rs2[4:0]

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode

R-type

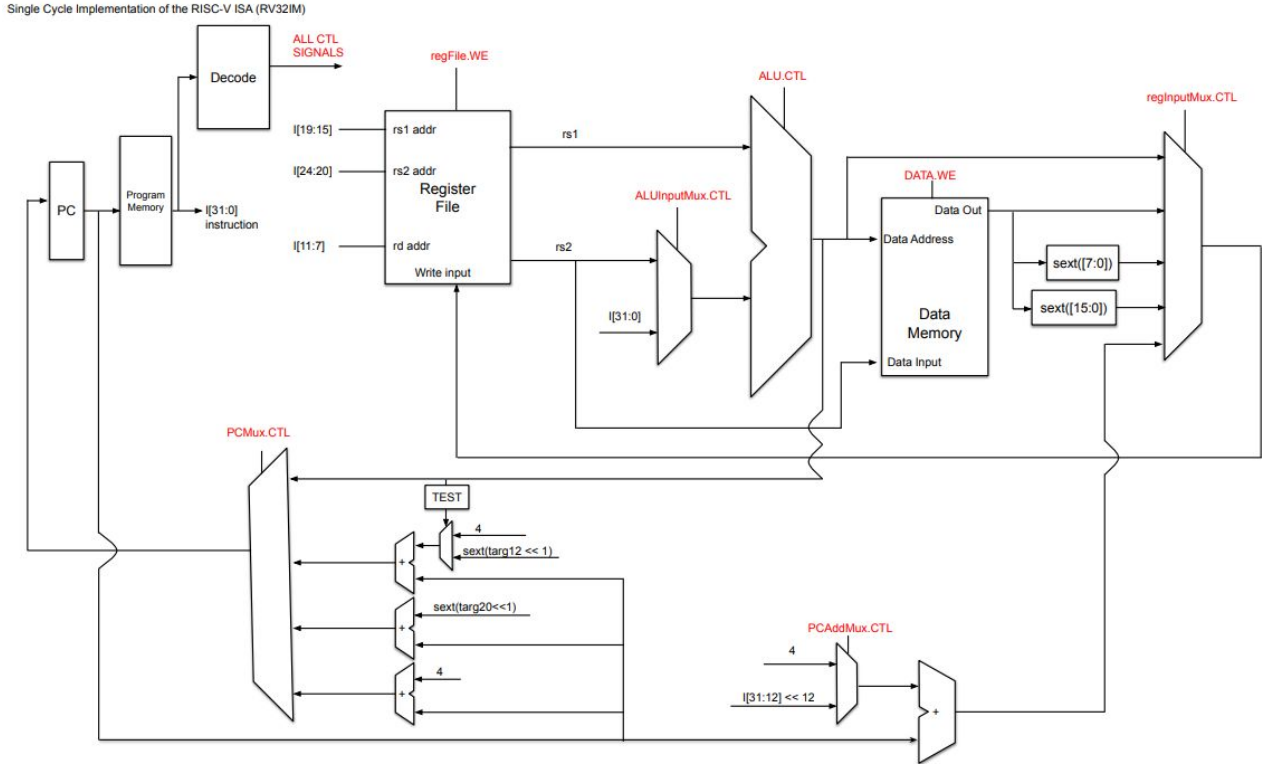


Example 3

instruction	fmt	opcode	fun3	fun7	semantics
`sb rs2,imm12(rs1)`	S	23	000		`mem[rs1+se(imm12)][7:0] = rs2[7:0]`

31 25	24 20	19 15	14 12	11 7	6 0	
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

Control Signal	Value
regFile.WE	
regInputMux.CTL	
ALUInputMux.CTL	
ALU.CTL	
DATA.WE	
PCMux.CTL	
PCAddMux.CTL	

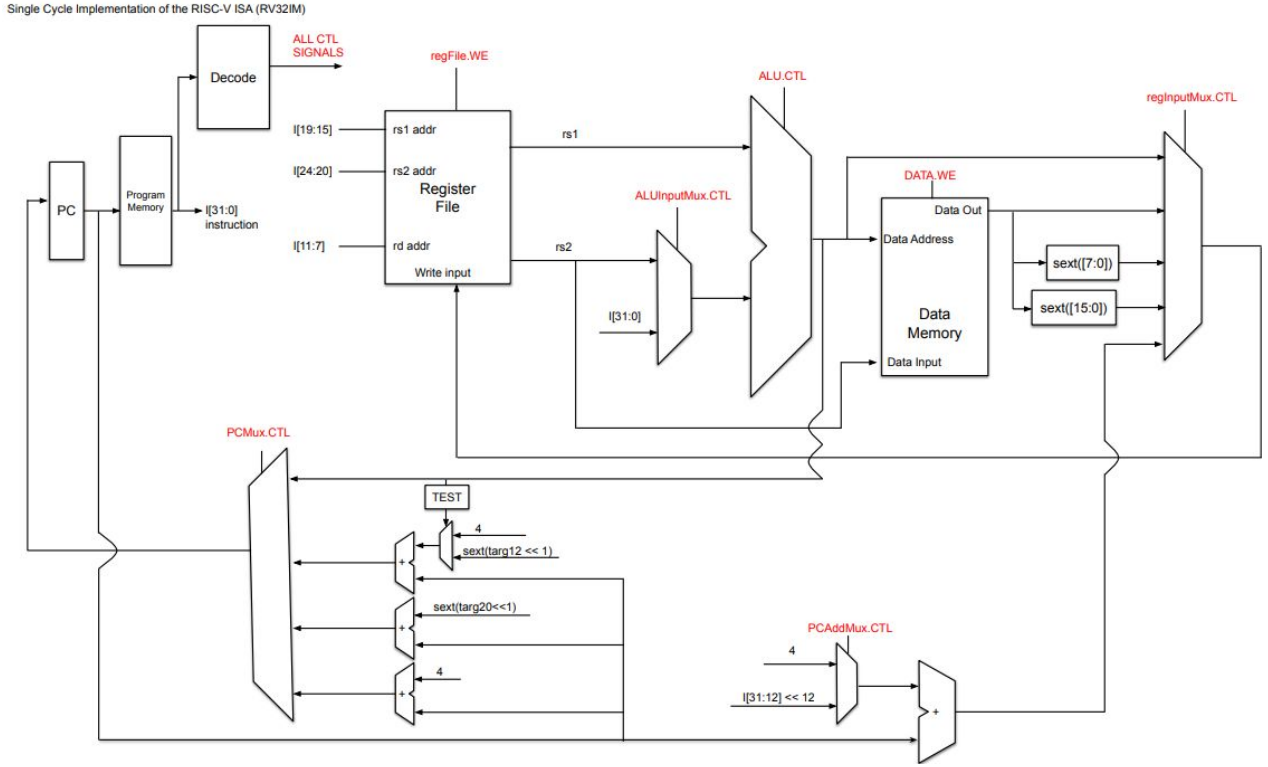


Sol 3

instruction	fmt	opcode	fun3	fun7	semantics
`sb rs2,imm12(rs1)`	S	23	000		`mem[rs1+se(imm12)][7:0] = rs2[7:0]`

31 25	24 20	19 15	14 12	11 7	6 0	
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

Control Signal	Value
regFile.WE	0
regInputMux.CTL	X
ALUInputMux.CTL	1
ALU.CTL	36
DATA.WE	1
PCMux.CTL	3
PCAddMux.CTL	X

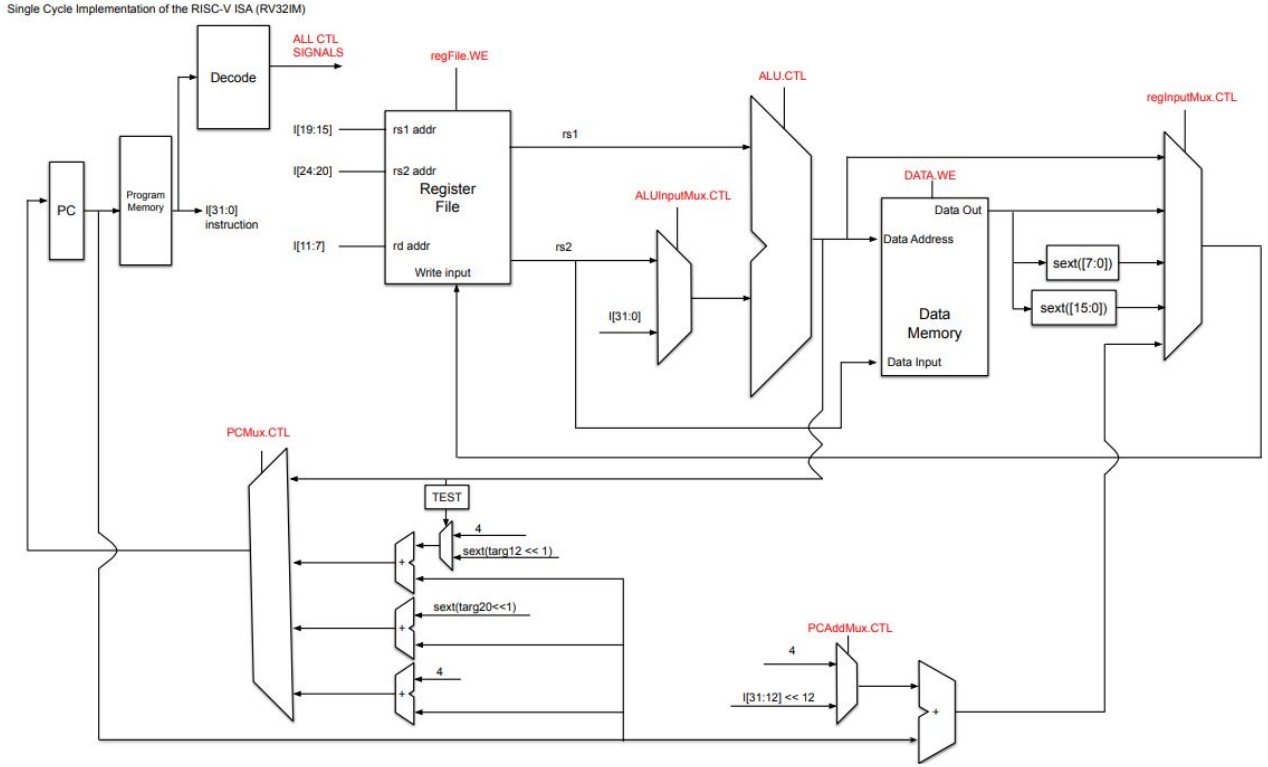


Example 4

instruction	fmt	opcode	fun3	fun7	semantics
bge rs1,rs2,targ12	B	63	101		if (rs1 ≥signed rs2) pc += se(targ12<<1)

31 25	24 20	19 15	14 12	11 7	6 0
imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode	B-type

Control Signal	Value
regFile.WE	
regInputMux.CTL	
ALUInputMux.CTL	
ALU.CTL	
DATA.WE	
PCMux.CTL	
PCAddMux.CTL	



Sol 4

instruction	fmt	opcode	fun3	fun7	semantics
bge rs1,rs2,targ12	B	63	101		if (rs1 ≥signed rs2) pc += se(targ12<<1)

31 25	24 20	19 15	14 12	11 7	6 0
imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode	B-type

Control Signal	Value
regFile.WE	0
regInputMux.CTL	X
ALUInputMux.CTL	0
ALU.CTL	22
DATA.WE	0
PCMux.CTL	1
PCAddMux.CTL	X