Lab 2: Assembly Language/Cpusim Introduction

Introduction to Assembly:

Assembly language is a programming language that is one step away from machine language. Typically, each assembly language instruction is translated into one machine instruction by the assembler. Assembly language is hardware dependent, with a different assembly language for each type of processor. In particular, assembly language instructions can make reference to specific registers in the processor, include all of the opcodes of the processor, and reflect the bit length of the various registers of the processor and operands of the machine language. An assembly language programmer must therefore understand the computer's architecture.

A statement in a typical assembly language has the form shown in Figure B.1. It consists of four elements: label, mnemonic, operand, and comment.

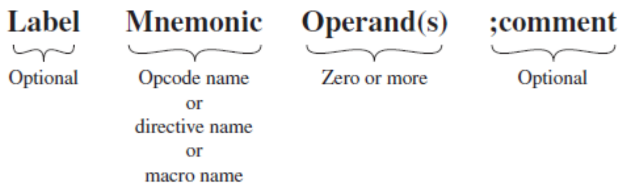


Figure B.1 Assembly-Language Statement Structure

OPERAND(S) An assembly language statement includes zero or more operands. Each operand identifies an immediate value, a register value, or a memory location.

As an example, here is a program fragment:

```
L2: SUB  EAX, EDX   ;subtract contents of register EDX from
                    ;contents of EAX and store result in EAX
    JG   L2         ;jump to L2 if result of subtraction is
                    ;positive
```

The minimal needed registers for operation of a basic computer:

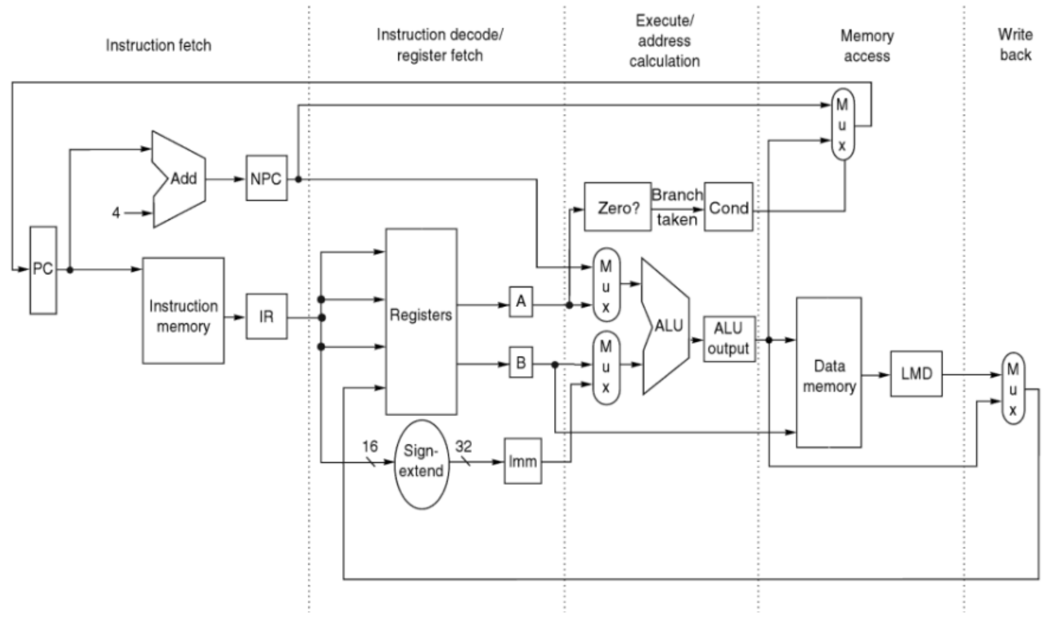
Table 1. List of Registers for the Basic Computer

Notations of Registers	Size (Number of bits)	Register Name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator register	Processor register
IR	16	Instruction register	Holds binary coded form of instruction
PC	12	Program counter	Holds address of a recent instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

The manner in which the processor executes an instruction and advances its program counters is as follows:

1. execute the instruction at *PC*
2. copy *nPC* to *PC*
3. add 4 or the branch offset to *nPC*

```
void advance_pc (SWORD offset)
{
    PC  =  nPC;
   nPC  += offset;
}
```



All instructions follow the same sequence of five pipeline stages:

1. Instruction fetch (IF): The instruction is fetched from memory and placed in the instruction register (IR).

2. Instruction decode (ID): The bits of the instruction are decoded into control signals. Operands are moved from registers or immediate fields to working registers. For branch instructions, the branch condition is tested and the branch address computed.

3. Execution (EX): The instruction is executed. Specifically, if the instruction is an arithmetic or logical operation, its results are computed. If it is a load-store instruction, the address is computed. All this is done by an elaborate logic circuit called the arithmetic-logical unit (ALU).

4. Memory read/write (ME): If the instruction is a load-store, the memory is read or written.

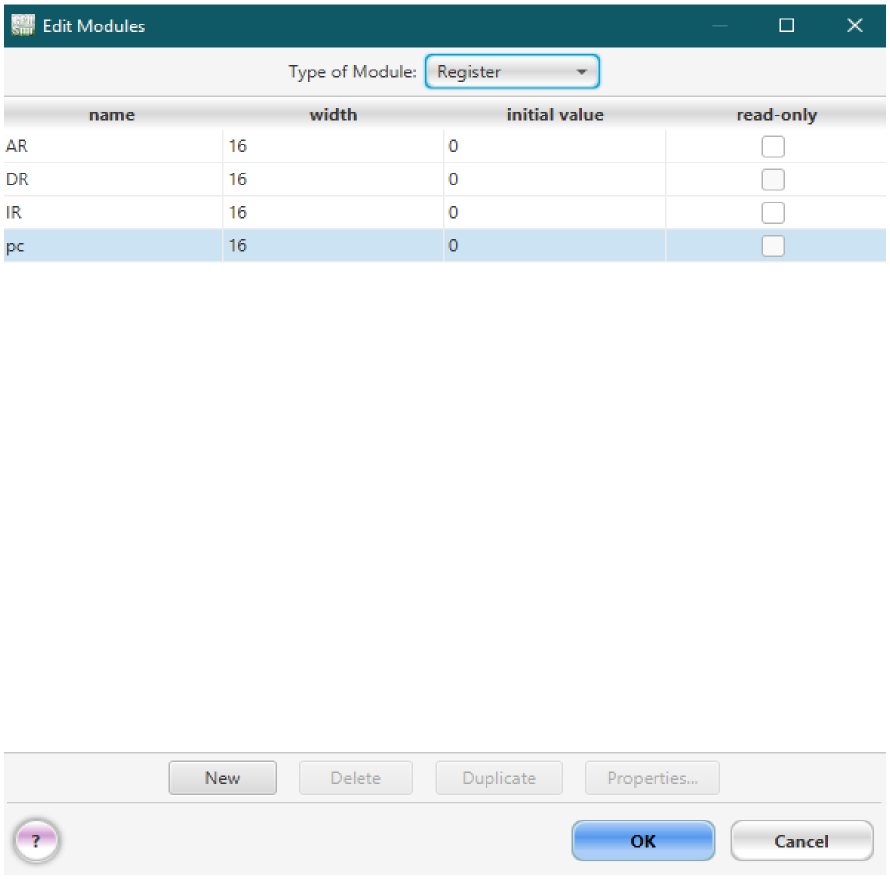
5. Write back (WB): The results of the operation are written to the destination register.

CPUSIM Example: The Fetch Sequence.

Steps for setting the fetch sequence in your machine:

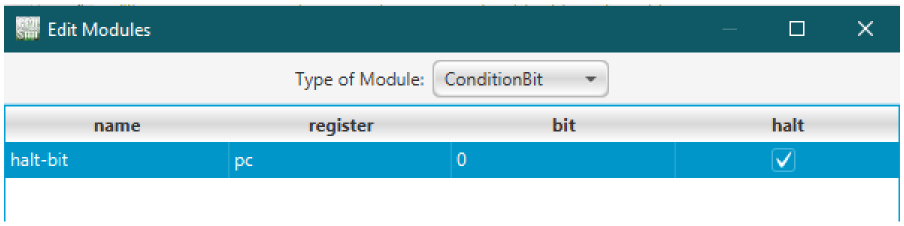
1- Set a fixed size for all your instructions (Example: 16 bits = 2 bytes)
2- Have a halt-bit on the PC register.
3- Set the needed micro-instructions.
4- Set the fetch sequence.

Go to hardware modules (Ctrl+k) and add PC, DR, IR and AR registers then set a halt-bit to the PC.



Type of Module: Register

name	width	initial value	read-only
AR	16	0	☐
DR	16	0	☐
IR	16	0	☐
pc	16	0	☐

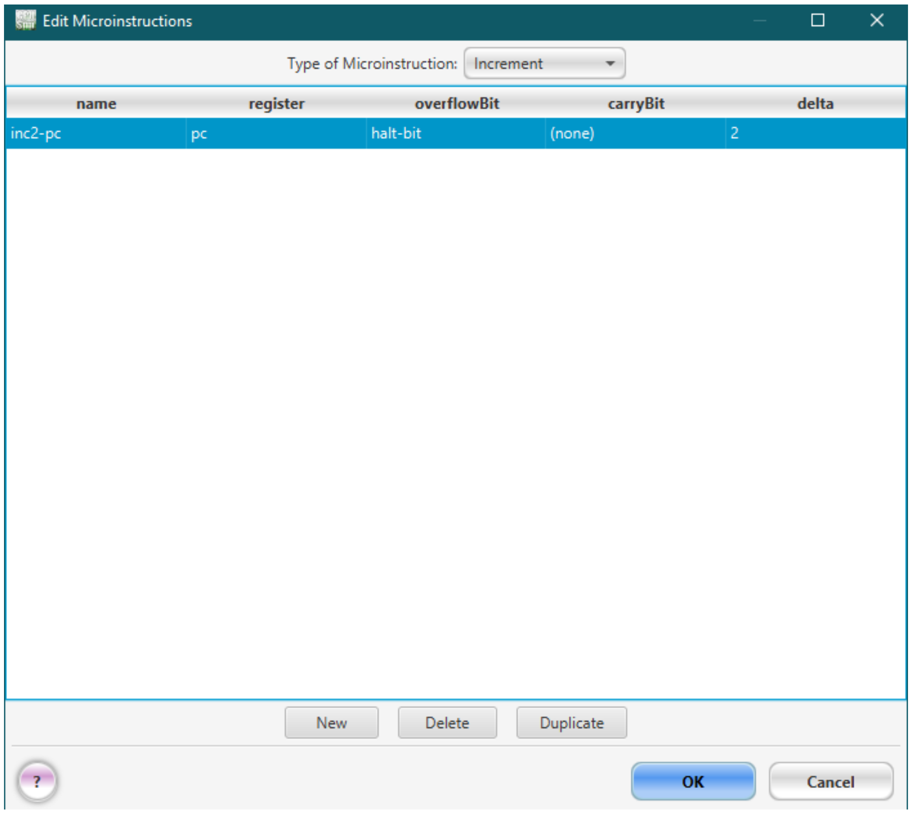


Type of Module: ConditionBit

name	register	bit	halt
halt-bit	pc	0	✓

Check the following manual for cpusim:
http://academicscience.co.in/admin/resources/project/paper/f201708271503817715.pdf

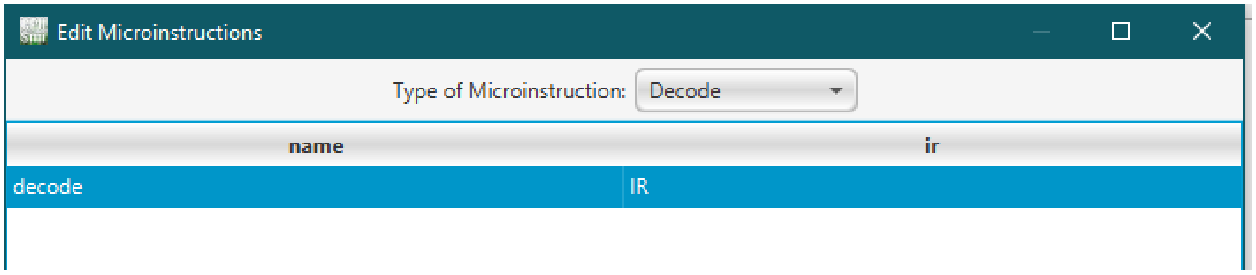
Go to microinstructions -> increment instructions then set an increment by 2 (the fixed size of all your instructions in bytes)



Type of Microinstruction: Increment

name	register	overflowBit	carryBit	delta
inc2-pc	pc	halt-bit	(none)	2

Then set the decode microinstruction which decodes and executes the instruction in the IR register.

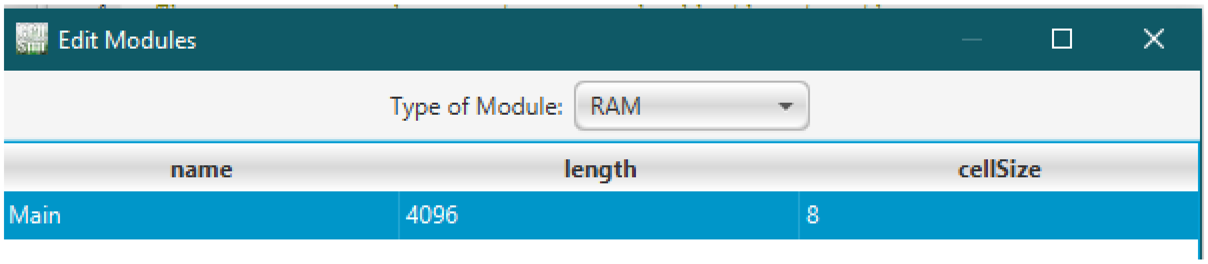


Type of Microinstruction: Decode

name	ir
decode	IR

Create a memory of size 4096 (Can be addressed by 12 bytes -> need Address Register AR to be 12 bytes long)

Go to hardware modules and create it there

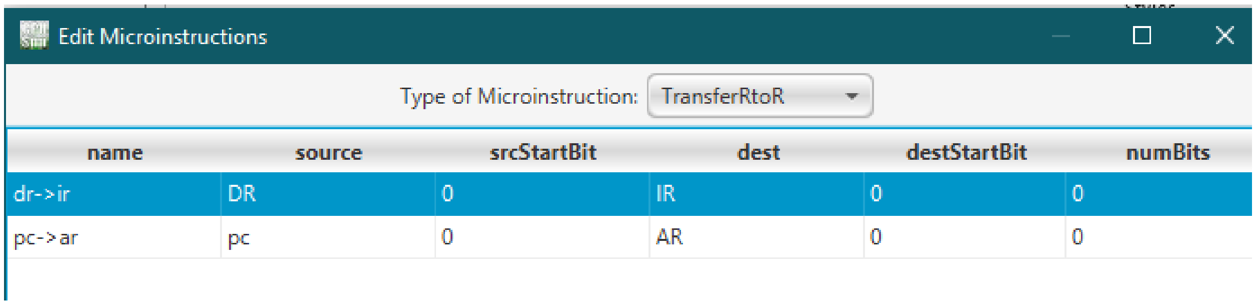


Type of Module: RAM

name	length	cellSize
Main	4096	8

Then create the microinstructions for performing the fetch sequence:-

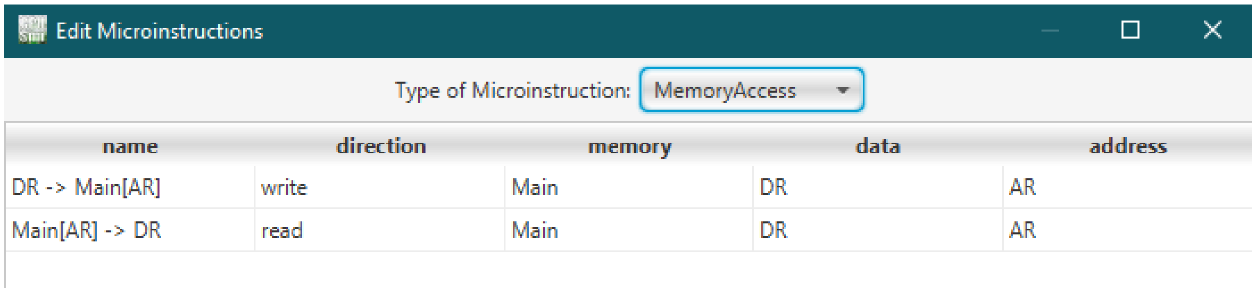
First the TransferRtoR type



Type of Microinstruction: TransferRtoR

name	source	srcStartBit	dest	destStartBit	numBits
dr->ir	DR	0	IR	0	0
pc->ar	pc	0	AR	0	0

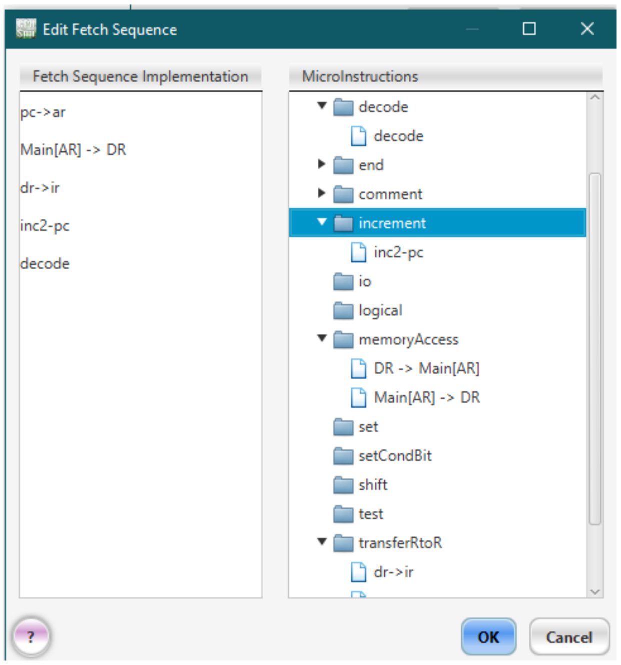
Then the Memory Access type



Type of Microinstruction: MemoryAccess

name	direction	memory	data	address
DR -> Main[AR]	write	Main	DR	AR
Main[AR] -> DR	read	Main	DR	AR

Then finish up by creating the fetch sequence and drag dropping the microinstructions you created.



Fetch Sequence Implementation

pc->ar
Main[AR] -> DR
dr->ir
inc2-pc
decode

Further reading (Check the fetch cycle): https://www.geeksforgeeks.org/different-instruction-cycles/

Example of an Assembly Architecture: MIPS machine.

ADD – *Add (with overflow)*

Description:	Adds two registers and stores the result in a register
Operation:	$d = $s + $t; advance_pc (4);
Syntax:	add $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 0000`

AND -- *Bitwise and*

Description:	Bitwise ands two registers and stores the result in a register
Operation:	$d = $s & $t; advance_pc (4);
Syntax:	and $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 0100`

BEQ -- *Branch on equal*

Description:	Branches if the two registers are equal
Operation:	if $s == $t advance_pc (offset << 2)); else advance_pc (4);
Syntax:	beq $s, $t, offset
Encoding:	`0001 00ss ssst tttt iiii iiii iiii iiii`

J -- *Jump*

Description:	Jumps to the calculated address
Operation:	PC = nPC; nPC = (PC & 0xf0000000) | (target << 2);
Syntax:	j target
Encoding:	`0000 10ii iiii iiii iiii iiii iiii iiii`

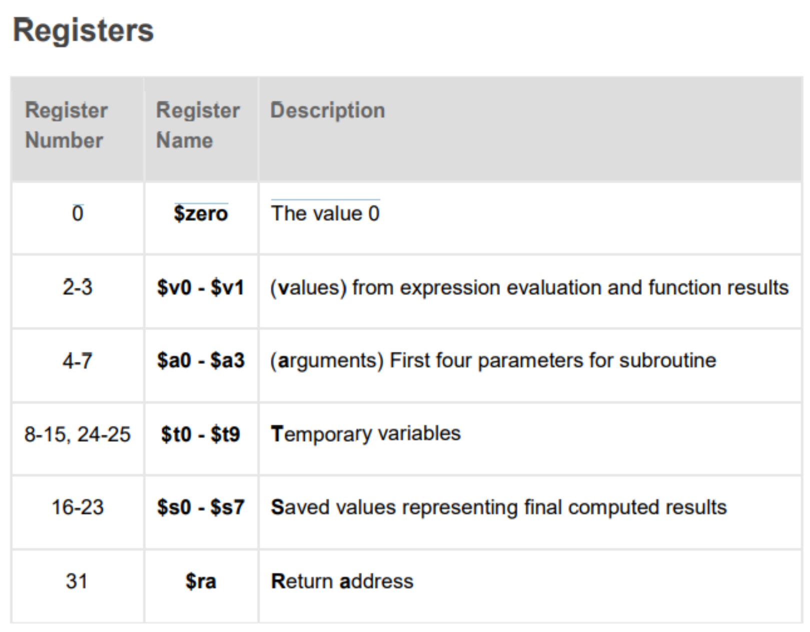
LW -- *Load word*

Description:	A word is loaded into a register from the specified address.
Operation:	$t = MEM[$s + offset]; advance_pc (4);
Syntax:	lw $t, offset($s)
Encoding:	`1000 11ss ssst tttt iiii iiii iiii iiii`

SW -- *Store word*

Description:	The contents of $t is stored at the specified address.
Operation:	MEM[$s + offset] = $t; advance_pc (4);
Syntax:	sw $t, offset($s)
Encoding:	`1010 11ss ssst tttt iiii iiii iiii iiii`

Registers in the MIPS architecture:



Registers

Register Number	Register Name	Description
0	**$zero**	The value 0
2-3	**$v0 - $v1**	(values) from expression evaluation and function results
4-7	**$a0 - $a3**	(arguments) First four parameters for subroutine
8-15, 24-25	**$t0 - $t9**	Temporary variables
16-23	**$s0 - $s7**	Saved values representing final computed results
31	**$ra**	Return address

Task:

You will be required to submit the following by next week's lecture.

- Create a new machine using CPUSIM.
- Add the basic registers: AR, DR, PC, IR, INPUT, OUTPUT, ACC.
- Add a 4096 byte memory.
- Create your fetch sequence.
- Add basic machine instructions: Load, Store, Add, Stop, Subtract, Jump.
- Create the corresponding micro-instructions for each of the machine instructions.

Next week we will see a few assembly examples and make our own on our created machines.