|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **ECE 350 Instruction Set Architecture** | | | | |  |
| **Instruction** | | **Opcode**  **(ALU Op)** | **Type** | **Usage** | **Operation** | |
| add | | 00000 (00000) | R | add $rd, $rs, $rt | $rd = $rs + $rt | |
| addi | | 00101 | I | addi $rd, $rs, N | $rd = $rs + N | |
| sub | | 00000 (00001) | R | sub$rd, $rs, $rt | $rd = $rs - $rt | |
| and | | 00000 (00010) | R | and $rd, $rs, $rt | $rd = $rs AND $rt | |
| or | | 00000 (00011) | R | or $rd, $rs, $rt | $rd = $rs OR $rt | |
| sll | | 00000 (00100) | R | sll $rd, $rs, shamt | $rd = $rs << shamt | |
| sra | | 00000 (00101) | R | sra $rd, $rs, shamt | $rd = $rs / 2^shamt | |
| mul | | 00000 (00110) | R | mul $rd, $rs, $rt | $rd = $rs \* $rt (32b X 32b); STATUS=1 if overflow | |
| div | | 00000 (00111) | R | div $rd, $rs, $rt | $rd = $rs / $rt (32b ÷ 32b); STATUS=1 if div0 | |
| CUSTOM R  (in ALU) | | 00000 (01000)  ...  00000 (11111) | R | CUSTOM\_R# $rd, $rs, $rt | $rd = CUSTOM#($rs, $rt) | |
|  |  |  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| j | 00001 | JI | j N | PC = N |
| bne | 00010 | I | bne $rd, $rs, N | if($rd != $rs) PC = PC+1+N |
| jal | 00011 | JI | jal N | $r31 = PC+1; PC=N |
| jr | 00100 | JII | jr $rd | PC = $rd |
| blt | 00110 | I | blt $rd, $rs, N | if($rd < $rs) PC=PC+1+N |
| bex | 10110 | JI | bex N | if(STATUS > 0) PC=PC+1+N |
| setx | 10101 | JI | setx N | STATUS = N |
| sw | 00111 | I | sw $rd, N($rs) | MEM[$rs + N] = $rd |
| lw | 01000 | I | lw $rd, N($rs) | $rd = MEM[$rs + N] |

|  |  |
| --- | --- |
| **Instruction Type** | **Instruction Format** |
| R | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Opcode [31:27] | RD [26:22] | RS [21:17] | RT [16:12] | shiftamt [11:7] | ALUop [6:2] | Zeros [1:0] | |
| I | |  |  |  |  | | --- | --- | --- | --- | | Opcode [31:27] | RD [26:22] | RS [21:17] | Immediate [16:0] | |
| J (I/II) | |  |  | | --- | --- | | Opcode [31:27] | Target [26:0] | | Opcode [31:27] | RD [26:22] | |

I-type immediate field [16:0] is signed 2’s complement and sign-extended to the full 32-bit word size.

J-type target field [26:0] is extended to the full 32-bit PC size using the upper bits from the current PC+1.

Register fields that are undefined are filled with zeroes by the assembler.

Register $r0 always equals zero. Registers $r1 through $r30 are general purpose. Register $r31 stores the

link address of a jump-and-link instruction.

Instructions that change control flow (beq, blt, j, jal, jr) do not have a delay slot.

Memory is word-addressed. The instruction and data memory address spaces are separate. Static data

begins at data memory address zero. Stack data begins at the end of the data memory and grows

downwards. There is no preset boundary between the end of static data and the start of the upwardsgrowing

heap; this is a property of the assembly program.

After a reset, all register values are zero and program execution begins from instruction memory address

zero. The memory's contents are not reset.

Useful hints for the Assembulator:

This assembly fragment (paste it into, and save- always save the file before assembling- the main window):

|  |  |
| --- | --- |
|  | .text  main: lw $r3, wow($r0)  lw $r4, wow($r0)  mul $r5, $r3, $r4  bex dead  addi $r7, $r0, 0x0FEEDF00  j quit  dead: addi $r7, $r0, 0x0DEADF00  quit: halt  .data  wow: .word 0x0000B504  mystring: .string ASDASDASDASDASDASD  var: .char Z  label: .char A  heapsize: .word 0x00000000  myheap: .word 0x00000000 |
| Comments | A simple program fragment, illustrating various assembly syntax for the Assembulator, which calculates the square of the value stored in location *"wow"* and sets the value of R7 to either 0x0FEEDF00 if the product doesn't overflow, otherwise 0x0DEADF00. |

Every program needs a ".text" and ".data" region, and a "main:" label. Program execution starts at "main:" Data variables can be allocated in the .data segment and referenced by their labels. For example, "mystring" is shorthand for the "location in memory of a .data labelled 'mystring'".

To assemble this fragment, enter the code (or open a file), save the code (to a file which INCLUDES a "." in it's name, like "mything.asm"), verify it (button with a check), and then you can simulate it (button with a play symbol). Next, you'll want to output some machine code and data in .mif files. Use the Assemble (or Assemble To) buttons (down arrows) to specify an output name (like "mything\_output") which will be used to create two files (dmem.mif and imem.mif). You can inspect these files from within the Assembulator, or add them to your project in Quartus (ultimately).

The assembulator also expands certain "pseudo opcodes" like this:

ldia = $rd, label + ldi $rd, N

ldi = $rd, N + addi $rd, $r0, N

ret = jr $ra

bgt = $rd, $rs, N + blt $rs, $rd, N

nop = add $r0, $r0, $r0

halt = j PC

It will also expand certain constants like:

$zero = $R0

$Ra = $R31

*Revised for ECE350, Spring 2015.*

CD.