****

**CS2610 - Assembly Language Programming**

Project Proposal:

**Assembler & ISA Design**

By **Aliasgar Musani (CS17B004)**

**Manish Kumar Srivastava (CS17B017)**

Dept. of Computer Science and Engineering

Indian Institute of Technology, Tirupati

Email: [cs17b004@iittp.ac.in](mailto:cs17b004@iittp.ac.in); [cs17b0017@iittp.ac.in](mailto:cs17b0011@iittp.ac.in)

**Objective:**

The major objective behind this project is to generate a new Instruction set and create an assembler for the same.

**The ISA:**

1. **Input File:**
   1. It has an extension of “.s”
   2. Each line consists of at most a label or an instruction.
   3. Contains only integer operations.
2. **Output File:**
   1. It has an extension of “.o”
   2. Executes only integer operations.
3. **Registers:**
   1. It consists of 32 general purpose registers and 6 special purpose registers.
   2. A is denoted by # followed by its number.
   3. SPRs:
      1. **#32:**

**HI Register**, contains the upper half of the number generated when two, 32 bit numbers are multiplied and remainder when they are divided.

* + 1. **#33:**

**LO Register,** contains the lower half of the number generated when two 32 numbers are multiplied and stores quotient when they are divided.

* + 1. **#34:**

**Zero register**, not accessible directly to anyone but can be used through instructions.

* + 1. **#IR:** Instruction register which stores the current instruction.
    2. **#PC:** Contains the address to the next instruction.
    3. **#MAR:** Stores the address of the memory to retrieve the data.
    4. **#MDR:** Stores the data fetched or to be written in the memory.

**Note**:

1. #32-#34 aren’t directly accessible to user but user can retrieve the data stored in it through instructions.
2. #PC, #IR, #MAR, #MDR are not at all accessible to a user and are a part of architecture.
   1. GPRs:
      1. **#0 - #9**:

10 **Temporary Registers** which can be used to store temporary and can be edited during a function call.

* + 1. **#10 - #17**:

8 **Save Registers** which won’t be modified by normal call and are used to save preserved data, need to make a different call.

* + 1. **#18 - #19**:

2 **Return Registers** which will contain return data from the function, which will be send by the function call return.

* + 1. **#20 - #23:**

4 **Argument Registers** which may contain your arguments forwarded to the function.

* + 1. **#24 - #25:**

2 **Kernel** **Registers** reserved for Kernel Calls.

* + 1. **#26 - #27:**

**Assembler Registers** use to process pseudo instructions. They can be used for normal processes but there may be chance of data loss if pseudo instruction is called in between.

* + 1. **#28:**

**Global Pointer Register** stores the address of the global label.

* + 1. **#29:**

**Stack Pointer Register** stores the address to the top of the data stack(negative addressed data can be done only on this, e.g. -8[#29], it points to the 3rd element in the stack).

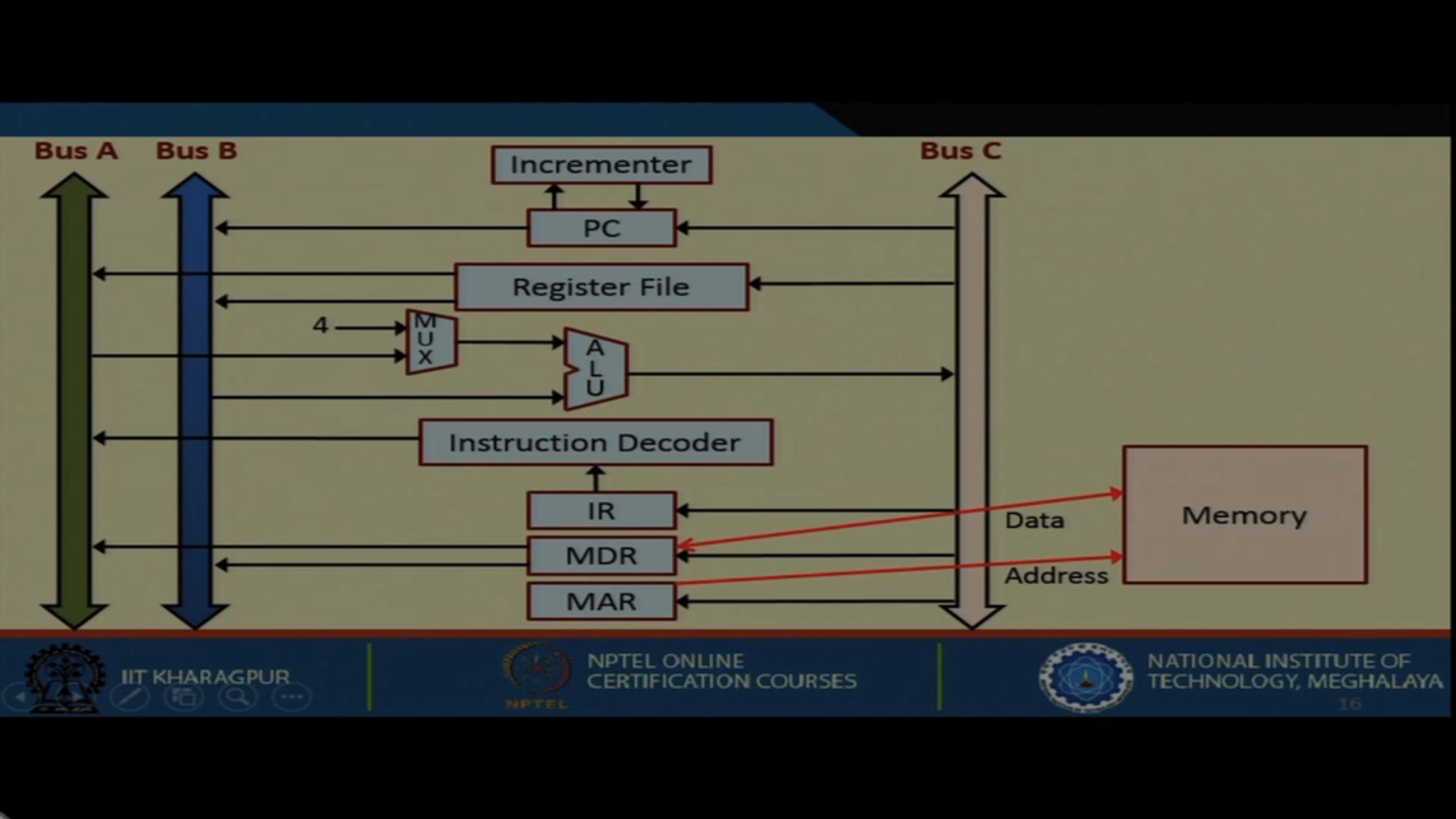
* + 1. **#30:**

**Frame Pointer Register** stores the address to the frame pointer.

* + 1. **#31:**

**Return Pointer Instruction** stores the return address to the original label, from where the current label was called.

1. **Word:**
   1. Word - size is 32 bit.
   2. System is byte addressable.
   3. Program Counter is Word Addressable.
2. **BUS:**
   1. 32 - Bit.
   2. 3 BUS system:



1. **Endian-ness:**
   1. Big Endian:
2. **Address:**
   1. Address size is 32 bits.
3. **Addressing Modes:**
   1. Register: Directly accessing the value in the register.
   2. Immediate: Taking the immediate integer
   3. Base plus offset: Base address plus some offset stored in another register

#1 = #1 + [#2]

New address is stored in #1. And it can be done using an add operation.

* 1. PC Relative: It’s Base addressing mode where base is replaced by PC

#PC = #PC + Offset.

* 1. Pseudo-Direct:

In this 26 bytes of address in jump are changed to original 28 bytes and then offset is produced.

All of the above ones are common, one addressing which I felt is missing and should be implemented is:

**Register Value Offset Indexing:**

**(#1)[#2] // New address is [#2] + #1**

This reduces one step of work in which new address had to be generated and had to be stored in another register using base addressing and then had to be used and again the incrementer was to be increased.

1. **Instruction Set:**
   1. **Instruction Specifications:**
      1. OPCODE size is 6 bytes
      2. Register Address Size is 5
   2. **R - Type:**
      1. In this type of instructions you will use add to move data.
      2. Will refer to #34 and take it’s help to move.
      3. OPCODE - 6 bits
      4. 3 Registers - 15 bits (src1, src2, dest)
      5. Shift Amount - 5 bits
      6. Function - 6 bits
      7. In case of using #34, we specify 3rd register all zeros and shift amount, all bits are 1.
      8. S registers can’t be directly accessed, you need to use special instruction use them, and only important should be stored in it.

|  |  |  |
| --- | --- | --- |
| Instruction | Action | Comments |
| ADD #1, #2, #3 | #1 = #2 + #3 | Can only change values in temporary and argument registers.  For unsigned ADDU |
| ADD #1, #2 | #1 = #2 + #34 | Used as a move instruction or to add zero to a number. |
| ADDS #10, #2, #3 | #10 = #2 + #3 | Used to store data in an S register only. |
| ADDS #10, #2 | #10 = #2 + #34 |  |
| SUB #1, #2, #3 | #1 = #2 - #3 | There are two types of functions in all, with two operands and three operands as in addition.  For unsigned SUBU |
| SUB #10, #2, #3 | #10 = #2 - #3 |  |
| MUL #1, #2, #3 | #1 = #2 x #3 | For unsigned MULU |
| MULS #10, #2, #3 | #10 = #2 x #3 |  |
| MULT #1, #2 |  | #32, #33 stores the upper and lower half of the number respectively.  For unsigned MULTU |
| DIV #1, #2, #3 | #1 = #2 / #3 | #32 stores the remainder, #33 stores the quotient, and not performed with zero.  For unsigned DIVU |
| AND #1, #2, #3 | #1 = #2 bit AND #3 | For both with two and three operands, all the following follow the same until mentioned. |
| ANDS #10, #2, #3 | #10 = #2 bit AND #3 |  |
| OR #1, #2, #3 | #1 = #2 bit OR #3 |  |
| ORS #10, #2, #3 | #10 = #2 bit OR #3 |  |
| XOR #1, #2, #3 | #1 = #2 bit XOR #3 |  |
| XORS #10, #2, #3 | #10 = #2 bit XOR #3 |  |
| NOT #1, #2 | #1 = bit NOT #2 |  |
| NOTS #10, #2 | #10 = bit NOT #2 |  |
| MFHI #1 | #1 = #32 |  |
| MFLO #2 | #1 = #33 |  |
| MFZ #1 | #1 = #xero |  |
| SLT #1, #2, #3 | If #2 < #3, #1 = 1  Else #1 = 0 | SLTU for unsigned. |
| LW #1, num[#2] | #1 = MEM(num + #2) | Similarly we have LB and LH, any GPR can be manipulated using this except $31. |
| SW #1, num[#2] | MEM(num + #2) = #1 | Similarly we have SB and SH. |
| LW #1, (#2)[#3] | #1 = MEM(#2 + #3) |  |
| SW #1, (#2)[#3] | MEM(#2 + #3) = #1 |  |

* 1. **I - Type:**
     1. Can not be used on S and other restricted registers.
     2. OPCODE - 6 bits
     3. 2 registers - 10 bits (Src1, Destination)
     4. Immediate Integer - 16 bits
     5. Pseudo for this type: LI #1, NUM - stores the 32 bit int into register 1

|  |  |  |
| --- | --- | --- |
| ADDI #1, #2, NUM | #1 = #2 + (16 bit) NUM | They can only be stored in temporary, no operation to store them in S type directly.  There is unsigned too ADDUI |
| MULI #1, #2, NUM | #1 = #2 x (16 bit)NUM | There is no division because may be if someone enters a zero as a bit number it may give errors, so trying to avoid direct integer division.  Unsigned MULUI |
| MULTI |  | Stored in HI and LO  Unsigned MULTUI |
| ANDI #1, #2, NUM | #1 = #2 bit AND NUM |  |
| ORI #1, #2, NUM | #1 = #2 bit OR NUM |  |
| XORI #1, #2, NUM | #1 = #2 bit XOR NUM |  |
| SLTI #1, #2, NUM | If #2 < NUM, #1 = 1  Else #2 = 0 | For unsigned SLTUI |
| LUI #1, NUM |  | Loads the num into upper 16 bits of the register |
| LLI #1, NUM | #1 = NUM | Lower 16 bits can be loaded |

* 1. **J - Type Instruction:**
     1. OPCODE - 6 bits
     2. Unconditional Branch:
        1. Address offset explicit - 26 bits, since memory is word addressable, therefore 28 bits.
     3. Conditional Branch:
        1. 2 registers - 10 bits (src1, src2)
        2. Address offset - 16 bits, but since memory is word addressable, therefore 18 bits.
     4. Pseudo for this type is: LA #1, address, this stores the address in the address label.

|  |  |  |
| --- | --- | --- |
| BEQ #1, #2, LABEL | If #1 == #2, then go to LABEL | LABEL contains an 16 bits offset |
| BNE #1, #2, LABEL | If #1 != #2, then go to LABEL |  |
| J LABEL | PC Jumps to the LABEL | LABEL contains 26 bits offset |
| JAL LABEL | #31 = #PC and then PC jumps to LABEL |  |

**Note:**

1. **.data, .text, .word, .space, etc.** are directly incorporated from mips, no change in their execution.
2. **Innovation:**
   1. ZERO register is made a special register instead of GPR.
   2. The extra register is used for assembler in case need to add more pseudo-instructions on a later stage.
   3. Added OFFSET indexing using the value stored in a register, which is missing in MIPS ISA.
   4. Made the usage of S register more safe as no one will be able to access it directly and so data stored in it will be more safe.