

Assembly Organization Project

***Developed by:***

*Abdulrahman Diaa 900162359*

*Yahia Farid 900161331*

***Presented to:***

*Dr. Tamer Elbatt*

Project Design

Firstly, we broke down the assembly file into two main objects: instructions and labels, each having its own struct which encompasses all its characteristics. Regarding the functions, we have 4 main functions that overarch everything. These include:

* Readfile(): This function opens the designated assembly file and loops over the file twice. For the first time, only the labels are checked. Their location with respect to the instructions is known and they are then pushed into a vector of labels for later use. For the second time, the instructions are taken. They are assembled (turned into machine code) which is then taken and saved appropriately into memory, using Big Endian.
* Run( ): This function goes through the memory and puts together the instruction (since memory is byte addressable). The machine code is then taken and parsed, meaning the bits are put into their respective positions such as funct7, registers, etc. Function execute is then called, which takes the instruction and performs the execution accordingly, modifying the registers or outputting onto the screen. These are then displayed using the Output() function.
* OutputOutputs(): This function goes through a vector of strings, which resembles any outputs made by the program using ECALL and displays them on screen at the very end.
* MipsConvert(): This function gathers the instructions from the memory, parses the machine code and then passes it to function FiveToMips(). This function then re-assigns the respective positions such as the registers and the opcode based on the different instruction formats.

Limitations

* Pseudo Instructions are not supported for mips conversion, even if they are native in Risc V.
* No AUIPC in mips.
* Program handles only up to section 2.6 in the Risc V manual.

Challenges

One of the biggest challenges we faced was that we originally did not handle negative offsets and immediates properly. This resulted in wrong outputs since the compiler used to read the offsets as positive. We then realized we had to sign extend the values to 32-bits, meaning we had to go back and modify the segment of our code which dealt with any offsets or immediates, which took up a lot of our time.

Assumptions

* SRA in risc-v supports variable, so we implemented it using SRA in mips loading the value from the register and loading it as SHAMT [allowing for what might have been SRAV in mips]
* Memory assumed to be so small that the JAL wouldn’t be affected by the 31st and 32nd bits of the PC.
* JALR in risc-v was converted to mips’ JALR. No JR is converted from any supported Risc-v instruction.
* Non-mips-native [pseudo] instructions are not supported for conversion.
* In LUI, mips format only takes 16 bits, so we decided to take the upper 16 bits of the LUI-RiscV immediate.
* Registers converted as follows:

|  |  |
| --- | --- |
| RISC-V | MIPS |
| X0 | $zero |
| X1 | $ra |
| X2 | $sp |
| X3 | $gp |
| X4 | $at |
| X5 | $k0 |
| X6 | $k1 |
| X7 | $t0 |
| X8 | $fp |
| X9 | $t1 |
| X10 | $v0 |
| X11 | $v1 |
| X12 | $a0 |
| X13 | $a1 |
| X14 | $a2 |
| X15 | $a3 |
| X16 | $t2 |
| X17 | $t3 |
| X18 | $s0 |
| X19 | $s1 |
| X20 | $s2 |
| X21 | $s3 |
| X22 | $s4 |
| X23 | $s5 |
| X24 | $s6 |
| X25 | $s7 |
| X26 | $t4 |
| X27 | $t5 |
| X28 | $t6 |
| X29 | $t7 |
| X30 | $t8 |
| X31 | $t9 |

How to use

After building and running the program, the user will be required to enter the filename which contains the assembly program (make sure it’s included in the project folder). Afterwards, the hexadecimal machine code will be output to the screen and to an external file in the format “filename\_Out.bin”. The user will be met with three different options:

* Pressing C for *continuous:* This means the code will execute fully and will only display the register states and any outputs made at the very end.
* Pressing N for *termination:* This means stopping the code and the program will not carry on executing.
* Any other key: This will execute the code instruction by instruction, allowing the user to check the register states after every instruction, as well as any other outputs.

Furthermore, after executing the main assembly code, the user will be asked to enter any key to have the hexadecimal machine code converted to hexadecimal MIPS machine code. This will be output to the screen as well as another output file in the format “filename\_Mips.bin”.