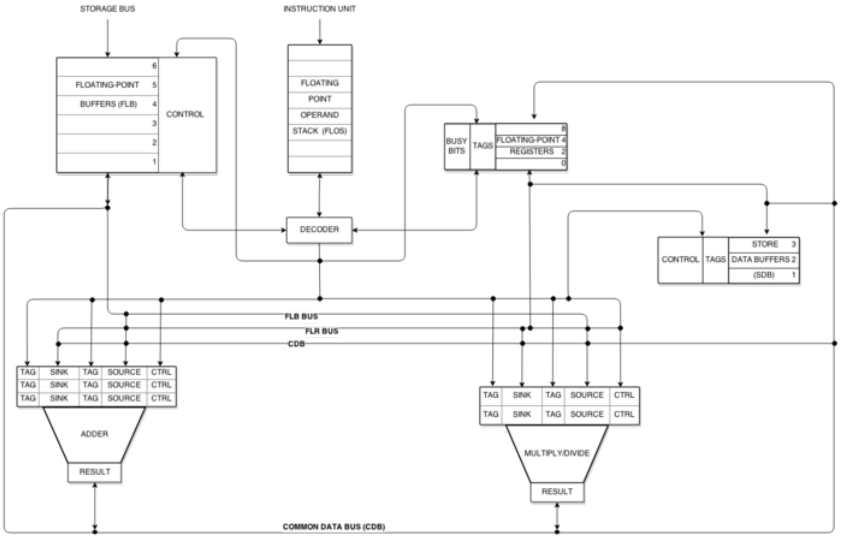
VL 803 - PROCESSOR ARCHITECTURE
FINAL PROJECT



Implementation of Tomasulo Out of Order Execution (Python)

K. Sailesh — IMT2017524
K. Prakash — IMT2017510

May 2020

Contents

1 TOMASULO

Tomasulo algorithm developed by Robert Tomasulo and first implemented in IBM System/360 facilitates for out of order instruction execution through implementation of additional hardware Reservation Station, ROB and Common Data Bus.

Steps of Tomasulo Algorithm:

1. Issue:

– Instruction Fetched and Decoded.
– Check if reservation station is free.
– Checks if the source operands will be produced by a current instruction.
– Issue instruction send operands (renames registers if needed).
– Monitor the CDB (common data bus) for operands → operand if available, is transferred to reservation station.

2. Execute:

– If source operands are ready then instruction sent to execution unit.

3. Write back and commit:

– Write on Common Data Bus to all awaiting units, register file and any waiting reservation station.
– Mark reservation station available after execution.
– Reorder buffer used to do in order commit

The RAW hazards(structural) are handled by hardware and WAW and WAR hazards are handled by register renaming.

2 IMPLEMENTATION

The following units/modules have been used in this implementation of Tomasulo. A lot of them have been written as objects making the system modular(variable co):

- **Clock**: Used for synchronization and timing. Mimics behaviour of a clock cycle. Has capability to show current clock cycle and increment the clock cycle.
- **Program Counter**: Mimics the behaviour of Program Counter by pointing to the address in Instruction Memory from where instruction has to be fetched. Has the capability to display the current address being fetched and update PC to PC+1 or any other address decoded from branch or jump instruction.
- **Reservation Station**: Each reservation station has the following fields -> type of the reservation station(Add/Mul); busy bit; operation to be performed on the data; source1; source2; data1; data2; ready bit; instruction number. In this implementation of Tomasulo 3 Add and 2 Mul reservation station units have been used. These numbers can be easily changed in initialization.py .

- **Load Station**: Each load station has the following fields -> busy bit; operation to calculate the address; source1 ;source2 ; immediate value; ready bit; instruction number. In this implementation 3 load station units have been used which can be changed in initialization.py
- **Store Station**: Each store station has the following fields -> busy bit; operation to calculate the address; source1 ;source2 ; immediate value; ready bit; instruction number. In this implementation 3 store station units have been used which can be changed in initialization.py
- All the three above modules have th functions to issue instruction to a reservation station unit of the instruction type if it is free and empty the reservation station after instruction has been issued for execution.
- **Adder**: Adder execution unit used to do addition and subtraction with given operands and produce result. Can be extended to do XOR; OR and AND.
- **Multiplier**: Multiplier execution unit used to do multiplication and division with given operands and produce result.
- **Common Data Bus**: On completion of execution of an instruction, publishes or broadcasts the data to waiting instructions in reservation, load and store stations and Reorder Buffer. The ROB entry is used as a tag to identify the variable.
- **Data Memory**: Implemented as a dictionary of Address and Value Mapping. Current usage of 4096 addresses. Can be varied as per requirement in the file data_memory.py. Data is loaded into this unit in initialization.py.
- **Architectural Registers**: Architectural Registers are considered a part of the state of a system and hence are changed only on correct execution(in order commit). Implemented as a dictionary of Register name (Rx) and Value Mapping. Current usage of 32 registers. Can be varied as per requirement in the file register_bank.py. Data is loaded into this unit in initialization.py.
- **Physical Register**: Physical Registers are used in register renaming technique which allows to prevent WAW and WAR hazards. Physical Registers have been implemented here as a dictionary of Physical register name (Fx) and value mappping. Current usage consists of 128 registers. Can be varied as per requirement in the file physical_registers.py. The Physical Register Map consists of the Mapping done between architectural and physical registers.
- **Instruction Memory**: Implemented as a dictionary of Address and Instruction Mapping. Current usage of 4096 addresses. Can be varied as per requirement in the file instruction_memory.py. Data is loaded into this unit in initialization.py.
- **Fetch**: Used to fetch the instruction at the address given by PC.

- **Decode**: Has the functions to decode an instruction into its constituent parts(Opcode, Operands); issue instructions to reservation station; and resolve branches and jumps.
- **Register Renaming**: Register renaming is a technique that abstracts logical registers from physical registers. This technique is used to avoid WAW and WAR hazards. The algorithm to do register renaming is as follows: Replace each source operand by the most recent value name in the designated register column. Then replace the destination operand by a new name and place the new name in the designated register column.
- **Reorder Buffer**: Reorder Buffer is a hardware component used to do inorder commit in an out of order execution processor. Reorder Buffer has been implemented as a dictionary of ROB column name (ROBx) to [instruction number,physical register,value] mapping. The functions ROBCommit() - write the value of physical registers into corresponding physical registers , GetFreeRob() and ROBClear() - clear the ROB entry after writting to architectural state are used in this implementation.
- **Instructions.txt**: Consists the format and list of instructions(RISC V ISA) which can be used to run the program in this implementation of Tomasulo.
- initialization.py contains all the function and variable initializations used in the main programs execution(maincopy.py)

3 EXECUTION

Instructions to run the Tomasulo demo :

1. Write the required program by converting to RISV ISA
2. The instructions need to be in the format mentioned in instructions.txt(adress of the instruction followed by the instruction itself in a single line sepeated by a comma)
3. The instructions are to be stored in a txt file say Programx.txt
4. Write the initial states of the data memory and register banks in seperate text files say RegBankx.txt and Memoryx.txt
5. The format of the above 2 files need to be of the form int a, int b where a represents the address/register number and b represents the value in it
6. Example txt files are available to see the correct format of files.
7. After creating the files, change the location to these files in initialization.py (the variables f1,f2,f3)
8. Execute the command python3 initialize.py
9. Now execute the command python3 maincopy.py which produces the output showing the out of order execution of the program and prints data corresponding to it

4 WORKING OF CODE

In initialization.py all the functions and variables being used in the program are imported and initialized. First we initialize the clock. Then, instructions and memory values are read from respective file and loaded into the Instruction and Memory units. Then we initiate the reservation, load and store stations and execution units corresponding to them. We initialize program counter to point to address 1(which can be changed to start reading instructions from a different location)

This file is then imported into maincopy.py(Our main file). Initially the decode unit would be free at the time of start. We start by fetching an instruction from the instruction memory pointed by PC. We decode this instruction and do register renaming and see if the instruction is a branch/jump or of other type. Branches and Jumps are resolved in the Decode stage(i.e. we wait if the variables are being generated by previous instructions and find the direction of execution and adrress of next instruction to be fetched). If the decoded instruction is of arithmetic type, we check for free reservation stations and issue the instruction into corresponding reservation station and also make an ROB entry.

Then if all the operands required for execution are available in a reservation station, we issue that instruction into execution unit and free the Reservation Station entry. Different Instructions have different latencies(i.e. completion time). If an execution unit is ready with output; We transmit this info on the Common Data Bus to the Reservation,Load and Store Stations and to ROB. Depending variables are replaced.

If the ROB head is available we start commiting to the main memory. Next instruction is fetched if the decode unit is free i.e. the previously decoded instruction is issued to a station. Else we wait for that decoded instruction to be issued and fetch the next instruction and follow the above process until we reach the END instruction

The final states of the memory; ROB; registers can be seen by uncommenting the required unit in the end of the program. To watch the changes uncomment/include the print unit comment in while(true) and while(len(earray)!=0) loops.

Changes can be made to the latencies of execution in CheckExec() function and the number of reservation/load store units can be varied by changing the loop count where ever iterating through the stations. Appropriate number of units are to be first initialized and added to RSC/LSC/SSC. The variable RSC is a combination of Adds and Mul units. Index has to be taken care of.