SPSIM: SuperScalar Processor SIMulater

CS305 Project Report

Yash Shah (160050002), Naman Jain (160050025), Utkarsh Gupta (160050032),
Rupesh (160050042), Sharvik Mital (160050059)

November 2018

Abstract

The aim of this project is to study the different techniques that are used to make the microprocessors execute programs fast, and test them out in practice by building a software simulator since none of the techniques give any useful theoretical bounds whatsoever. In this pursuit we first need to understand how we can make a program execute faster. Surely, we could do so by improving the algorithm, but what about problems for which there is no efficient algorithm known to exist (NP-hard problems)? Clearly, the only way to run a given program faster in such a scenario is to make changes in the mechanisms and policies of the system on which it is being executed. Where do the possibilities of optimization lie? To answer this question, we need to ask ourselves what the bottleneck for execution of the program is at the hardware or architectural level. There has been a lot of effort put into solving this question, and we already know a few major optimizations like pipelining and forwarding which have been implemented in the architecture of microprocessors to make them execute programs faster. In this project, we look at optimizations that came later and develop a superscalar processor simulator which makes use of branch prediction techniques and out-of-order execution (or dynamic scheduling) to increase the throughput of the processor.

1 Problem Statement

To help us focus on the essence of the problem and not on all the system level details, we model a program as a sequence of instructions that are executed "sequentially" by the microprocessor. Note that the sequential execution of instructions is the semantic and not the architectural requirement; we can execute instructions out of order as long as we maintain the correct semantics of the program.

1.1 Possible Directions

If we think about the bottleneck of program execution, we can come up with atleast the following three solutions:

1. ***Reduce instruction latency***: Reducing individual instruction latency reduces the execution time of the program since each individual instruction can be completed sooner.

2. ***Using more than one processors***: Using more than one processor to execute the program allows us to exploit parallelism in the program (if any), and can thus result in faster execution of the program.

3. ***Instruction level parallelism***: We can also execute a program faster if we can initiate and execute more than one instruction per clock cycle; this is referred to as instruction level parallelism or *superscalar processing* in the literature. This approach would require us to use extra hardware to execute multiple instructions simultaneously in the same stage of the pipeline in a clock cycle.

We will be focusing on the third approach in our project.

2 Superscalar Processing

Superscalar processing allows us to initiate more than one instructions simultaneously and execute them independently. We would like to point out the difference between superscalar processing of instructions and pipelining instructions. Pipelining allows execution of multiple instructions in a clock cycle but it requires them to be in different stages — superscalar processing, on the other hand, doesn't require so. Further, we cannot initiate multiple instructions in a pipeline, while we can do so in superscalar processing. This is made possible through addition of extra hardware.

We would ideally expect that if we are initiating two instructions per clock cycle, then our program execution would take half the number of clock cycles it was taking before. However, this is not always possible as discussed below:

2.1 Challenges in Superscalar Processing

Superscalar processing requires us to execute more than one instruction per clock cycle. But this is not as simple as it sounds, because more often than not those two consecutive instructions have some dependency between them i.e. there is often a *hazard* between two consecutive instructions which prevents them from being executing simultaneously. Thus, we need to detect and resolve such data hazards if possible, otherwise we won't be able to effectively utilize the extra hardware that we have put in the processor for executing multiple instructions in a clock cycle.

Another major obstacle are the branch instructions, where we do not know at the `IF` stage which instruction to fetch next. To minimize our losses in terms of clock cycles we use branch prediction techniques and do *speculative execution*.

3 Resolving Hazards

There are three types of hazards between instructions — **RAW** (Read-After-Write), **WAW** (Write-After-Write) and **WAR** (Write-After-Read). The manner in which they are handled is described below:

3.1 Resolving WAR/WAW hazards

In literature, WAR and WAW hazards are not considered "true hazards" (unlike RAW hazards); instead, they are considered "pseudo hazards". This is because **WAR and WAW hazards can always be removed by use of register renaming and remapping**. We explain this through an example.

Example 3.1 Consider the following assembly code snippet. Assume that there are 32 logical registers in this ISA, and 64 physical registers in hardware.

1. `mult $t3,$t2,$t1`
2. `add $t3,$t2,$t0`

There is WAW hazard between the two instructions if both instructions write to the same *physical* register, because in that case instruction 2 would have to wait for instruction 1 to write in order to maintain the correct semantics of the program. However, if we do dynamic mapping of logical registers with the physical registers for each instruction and store the mappings, then the second instruction can write to a register as soon as it has the result. This is because it is now writing to a different *physical* register, and the instruction that will read from `$t3` will read from this physical register since we have stored the mapping. Similarly, we can remove WAR hazards.

Thus through remapping of registers and appropriate state maintenance, we can remove WAR and WAW hazards.

3.2 Resolving RAW hazards

RAW hazards can not be removed by use of register remapping and renaming. Instead of trying to *remove* them, we try to utilize the inevitable stalls for executing other instructions by using *dynamic scheduling* or *out-of-order execution*. If a stage comes where we have to execute an instruction that has a RAW hazard with any of the ongoing instructions, instead of waiting we send such an instruction for execution (instead of the former instruction) which doesn't have a RAW hazard with the ongoing instructions. Such an instruction might not exist, in which case there would be a stall in the pipeline.

4 Branch Prediction

We use dynamic branch prediction based on the computation history of the program to predict what the next instruction fetched should be. For dynamic branch prediction, we use a two-bit branch predictor and a table data structure which tells the simulator the predicted address the branch should go to. The initial predicted *jump* address for all instructions is assumed to be the next instruction. If we encounter a branch instruction for the first time, it is assumed to be in the `STRONGLY_NOT_TAKEN` state (refer figure 1). Otherwise, we use some bits from the program counter of the branch instruction to index into the table and retrieve state of the branch instruction and the predicted jump address. On each successful prediction and misprediction we check and update the corresponding entry of the branch instruction in the table, if necessary. Figure 1 is the transition diagram of the finite state machine that represents the execution of our two-bit branch predictor.

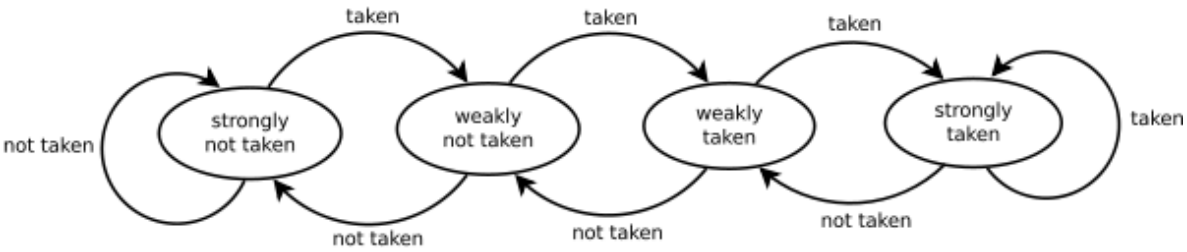

Figure 1: 2-bit branch predictor used in SPSIM

5 Implementation Details

In this section we describe implementation level details of the simulator that we designed and implemented. Accompanying code for the same can be found at `https://github.com/ys1998/spsim`.

5.1 Simplifying assumptions

Developing a general superscalar processor simulator which works for a large set of operations is a difficult task — one which we are currently unable to accomplish due to the time constraint. However, we have made certain simplifying assumptions that have made simulation of a subset of those operations possible. We describe those assumptions here:

- We do not model the behaviour of cache in SPSIM, i.e. we assume 100% hit rate. This is because it is not required for our objective and would have only complicated the implementation.
- We assume 32 bit instructions and follow the MIPS ISA as taught in class.
- For now, we consider a very limited set of instructions in SPSIM. We don't consider floating point operations or *pseudo* instructions. We consider only `add, sub, mult, div, bne, beq, sw, lw` operations.
- Our ISA has `mult` instruction of the format `mult rd,rs,rt` instead of the MIPS format.
- Sizes of buffers are considered to be unlimited; this allows us not to worry about stalls in the pipeline arising because of full buffers.
- We do not implement forwarding; we, however, do implement reading and writing of registers in the same cycle — write in the first half and read in the second.

5.2 Challenges overcome

1. **Flushing invalid instructions from the pipeline on branch misprediction**

   Whenever a branching instruction such as `beq` or `bne` is encountered, we predict the outcome (i.e. whether the branch will be taken, and the target address if it is taken) in the `DE` stage itself in order to avoid stalls. This prediction could be wrong sometimes, and the correct result is known only when the condition is actually computed in ; in such a situation, we need to *restore* the pipeline state to that before any speculation was made. Since register renaming is done only for the destination register (`rd`), we store the previous mapping value (-1 if no mapping existed earlier) for every instruction (as `_rd`). We also maintain a unique identifier for each instruction (`ID`), which grows monotonically in value with total number of instructions. Each category of operations is handled in a slightly different manner:

   - ***Integer operations*** : Each clocked entity has a `flush()` method that clears/removes all instructions with `ID` greater than that of the branch instruction from all of its input, output and internal buffers (buffers wherein `ID` is still unassigned are cleared completely). Removal of instructions from the *ActiveList* is done a little more cautiously since we need to restore the original mapping of `rd` correctly. For this, instructions are accessed in the decreasing order of their `ID`s, mapping of `rd` is replaced by that of `_rd`, the *BusyBitTable* is updated and finally the instruction is removed.

- `lw` ***operation*** : This instruction is dealt in a similar manner as other *integer operations*. This is because here too we need to restore the register mapping — we are not concerned about the value in the register since all later reads are going to be invalidated anyways, and the original value in memory remains unaffected.
- `sw` ***operation*** : These instructions were trickier to handle. This is because once they complete execution, the original value in memory is overwritten and cannot be restored if we realize later that the instruction was wrongly speculated. In order to prevent this from happening, we issue `sw` instructions for execution only when there are no other active branch instructions *ahead* of it in the *ActiveList* i.e. a store instruction is executed only when it is certain that it will not be flushed. Thus, when a `sw` instruction is speculated on encountering a branch, it will always be placed *behind* the branch instruction in the *ActiveList* and will be executed only after the former safely graduates. Thus, flushing of `sw` instructions simply involves scanning the *AddressQueue* and *ActiveList* and removing those instructions whose `ID` is greater than that of the branch instruction.

2. **Handling memory operations/instructions**

   Memory instructions have both an `EXEC` stage and a `MEM` stage and we have to deal with both of these operations sequentially. Also, out of order execution in case of load & store instructions is not trivial because instructions may have dependencies in memory addresses which have to handled separately (since the address space can be huge, storing a busy bit per word is not a solution). What makes matters worse is that these dependencies can only be found at run-time since addresses have to be computed and these calculations might themselves have dependencies. Note that dependencies till the `EXEC` stage are handled similarly to those of integer operations; however, issuing to `MEM` stage is non-trivial.

   The simplest solution one can obey is serializing memory accesses, but this defeats the purpose of *out-of-order* execution. Instead, we optimize these operations in the following manner:

   - If there are only load operations in the pipeline we forward them for address calculation and memory fetch.
   - If a store operation is present in the pipeline, and it is stalled due to an uncalculated memory address, then no further `sw` or `lw` instruction with higher `IDs'` can be issued to the `MEM` unit.
   - However, if the `sw` operation is stalled only due to a busy *target* register (i.e. the register which will store the value at the memory address), we can bypass it and check for later `lw`/`sw` operations that **do not** have a memory dependency with the former `sw` operations (i.e. their target addresses are different). In case such an operation is found, it can be sent to `MEM` stage.

3. **Simulating multi-cycle execution stage instructions**

   We need to be careful while executing multi-cycle execution stage instructions such as `mult`. This is because if we are not, then the latch storing the next instruction to be fed into the execution stage will get overwritten without getting read by the ALU, resulting in the loss of instructions and hence incorrect execution. To resolve this problem, we do the following: if a latch has not been read in a clock cycle, all the clocked entities (refer figure 2) behind that latch in the datapath get stalled, and thus the instruction in the latch is preserved. When it is picked up by the ALU after it completes its execution, the clocked entities start functioning again.

4. **Handling nested branch instructions**

   Nested branching can arise in many common scenarios and hence is of significant concern. It might seem that speculating the results of two (or more) branch conditions and accordingly updating the program flow might lead to incorrect execution. However, following the aforementioned approach of using `ID`s for flushing invalid instructions on misprediction leads to correct results. Although speculation might happen in either direction (increasing or decreasing value of the program counter, `PC`), restoring the original state of registers simply involves back-tracking `rd`'s mapping in decreasing order of `ID`s which is irrespective of branch prediction.

5. **Determining program termination**

   We determine termination of the program when the *ActiveList* (refer figure 2) becomes empty. This works because the *ActiveList* stores all active instructions, and when it becomes empty it indicates that all the instructions have been executed and have retired/graduated.

5.3 Pipelining in SPSIM

In the current implementation, each instruction passes through a subset of the following **seven stages**:

1. `IF` or **Instruction Fetch** - instructions fetched from I-Cache
2. `DE` or **Decode** - instructions decoded; register mapping and renaming
3. `RF1` or **Register Fetch, substage 1** - operand registers acquired; out-of-order scheduling done
4. `EXEC` or **Execution** - operation performed (latencies can be > 1)
5. `RF2` or **Register Fetch, substage 2** - source/destination registers acquired for *lw/sw* instructions respectively; computed memory address also fetched from ALU3
6. `MEM` or **Memory** - reads/writes from/to D-Cache performed
7. `WB` or **Write-Back** - result of operation written to destination register; instruction graduation

Integer arithmetic instructions use the `IF`, `DE`, `RF1`, `EXEC` and `WB` stages. Note that some operations such as `mult` and `div` have a multi-cycle `EXEC` stage.

Load Store instructions use `IF`, `DE`, `RF1`, `EXEC`, `RF2`, `MEM` and `WB` stages. They are added in *AddressQueue* in `DE` stage. Address is calculated in `EXEC` stage and instruction is sent to `MEM` stage later according to the logic mentioned in Section 5.2.

For branch instructions such as `beq` and `bne`, prediction is performed in the `DE` stage and hence the speculated instructions are fetched after this stage. The condition is actually evaluated only in the `EXEC` stage. In case of misprediction, the correct instructions will start executing only *after* the `EXEC` stage has completed. Thus, a branch instruction also passes through `IF`, `DE`, `RF1`, `EXEC` and `WB` stages.

5.4 SPSIM Logical View

Each clocked hardware component is modelled using an appropriate derived class of the more abstract `ClockedEntity` class. Each of these entities has a `tick()`, `tock()` and `flush()` method, which are used to perform necessary actions during the events of a rising clock edge, falling clock edge and flushing of invalid instructions respectively. The various *clocked entities* used are as follows:

1. `Fetcher` - It fetches two instructions every cycle using the current value of program counter from the I-Cache and pushes them to an intermediate buffer between itself and the decoder. It responds to flushing by correcting the program counter value to the one before speculation.

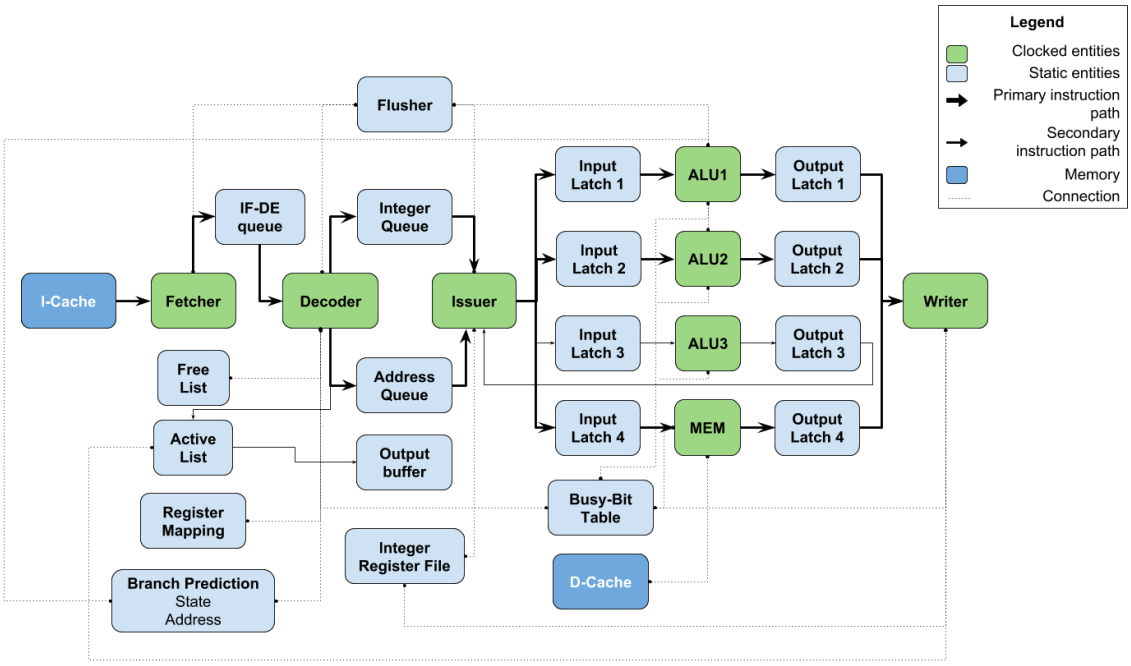

Figure 2: SPSIM Datapath

2. `Decoder` - It retrieves two instructions every cycle from the buffer and decodes them. Decoding involves logical-to-physical register mapping and renaming using `FreeList` and `RegisterMapping`, and also updating the `BusyBitTable` for handling dependencies. After this, instructions are pushed either to the `IntegerQueue` or the `AddressQueue` depending on their type (arithmetic / branching or load/store respectively).

3. `Issuer` - It is responsible for out-of-order issuing of instructions to the ALUs. In every cycle, it issues one instruction per ALU (if it is free) by checking whether operands for that instruction are 'available'. Each ALU can handle only a subset of possible operations, and the Issuer ensures that each instruction is passed to the input latch of the correct ALU.

4. `ALU1` - It handles operations such as `add`, `sub`, `bne`, and `beq` with higher priority towards branching instructions. It reads from an input latch, and writes to an output latch. It is stalled whenever either of them isn't free (i.e. there is no instruction to read from input latch or the previous output has still not been read from the output latch).

5. `ALU2` - It functions similar to `ALU1`, but handles `mult`, `div`, `add` and `sub` operations with higher preference towards `mult` and `div`.

6. `ALU3` - It functions similar to `ALU1` & `ALU2`, but handles only `add` operations for computing the memory addresses for `lw` & `sw` instructions.

7. `MEM` - It is used by `lw` and `sw` instructions to communicate with the memory i.e. perform reads from and writes to specified addresses. Note that the target address needs to be computed

beforehand and fed to this unit.

8. `Writer` - It reads the outputs from all three latches (i.e. the output latches of `ALU1`, `ALU2` and `MEM`) and performs writes to the corresponding destination registers. It also graduates completed instructions from the `ActiveList`.

9. `Flusher` - In case of branch misprediction, it flushes all invalid instructions from the pipeline and restores the original state by updating the program counter and reverting back altered logical-to-physical register mappings.

Apart from these clocked entities, several *static entities* were also used, which are briefly described below:

1. `ICache`, `DCache` - simulate the instruction and data caches respectively
2. `Buffer` - used for modelling intermediate and internal storage units between/within clocked entities
3. `BusyBitTable` - a per-physical-register table indicating whether that register is busy or not (i.e. whether it is in the process of being written to or not)
4. `RegisterMapping` - stores the most recent physical register that is mapped to a given logical one
5. `FreeList` - maintains a constantly updated list of physical registers that are free (i.e. they don't store any valuable information, and hence can be written to)
6. `ActiveList` - list of all instructions active in the pipeline; instructions are removed once they graduate
7. `IntegerRegisterFile` - the group of physical registers
8. `IntegerQueue` - stores instructions corresponding to integer operations
9. `AddressQueue` - stores instructions corresponding to memory operations
10. `Latch` - temporarily holds a value and prevents writes until the previous value has been read
11. `BranchPredict`, `BranchPredictAddr` - buffers used for storing auxiliary information for branch prediction i.e. current state in the prediction FSM (see figure 1), and predicted address for a given program counter respectively.

The connections between these static and clocked components are modelled using pointers. When a component is instantiated, it is provided with the pointers to those entities with which it is connected — either for reading input or writing/pushing output.

6 Examples

In this section we present some code snippets and the corresponding output of the simulator, and point out the salient features of our implementation.

Ex 1: Demonstrating out-of-order execution, hazard resolution and multi-cycle execution
In figure 3, we have the following hazards:

1. **WAW hazard** between instruction 1 and instruction 2. Since we are resolving WAW hazards using register remapping, instruction 2 doesn't have to wait for instruction 1 to complete execution. It is worth pointing out that multiple cycles were saved (not just 1) and if the program

Figure 3: Example 1

were only two instructions long, the effective time of execution would be just the time to execute the first instruction.

2. **RAW hazard** between instructions 2 and 3 & between instructions 3 and 4. Our strategy to deal with RAW hazards was to execute instructions out-of-order, and as can be seen in the figure, instruction 5 enters `EXEC` stage before instruction 4 and at the same time as instruction 3.

3. Note the multi-cycle `EXEC` stage of `mult` instruction.

Ex 2: Demonstrating branch prediction and branching

In figure 4, we observe the execution of a `for` loop through `bne` statements in the following phases:

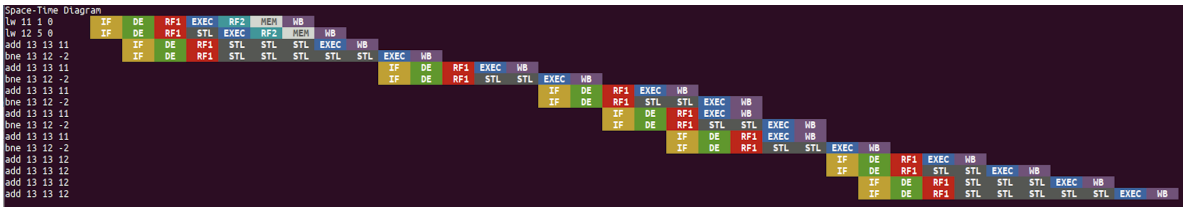

Figure 4: Example 2

1. The first instance of `bne` is predicted as *not taken* (which is the initial branch prediction state) and this misprediction results in a penalty of 4 cycles due to the final, correct condition evaluation in `EXEC` stage. There are some extra stalls due to `lw` instructions and RAW hazards.

2. The second instance also suffers the same number of stalls as the first one (except those caused by `lw` instruction), but the benefit is that due to the mispredictions for these two instances, the predictor for the corresponding program counter has moved to the *branch taken* state.

3. For all instances, starting from the third upto the penultimate one, the branch predictor correctly predicts the state of the branch as well as the target jump address. All this is achieved within a minimal delay of 1 cycle even when the true calculation is happening much later. Thus, prediction and speculative execution provides a huge improvement, since otherwise the branch instruction would have faced many stalls due to its dependency on the previous instruction.

4. The last "loop breaking" instance will again face 4 stalls due to misprediction since the predictor was expecting the loop to continue just like for the previous instance. Note that all invalid instructions that were a part of the speculation are safely cleared.

Ex 3: Demonstrating the scheme for load and store operations (1)

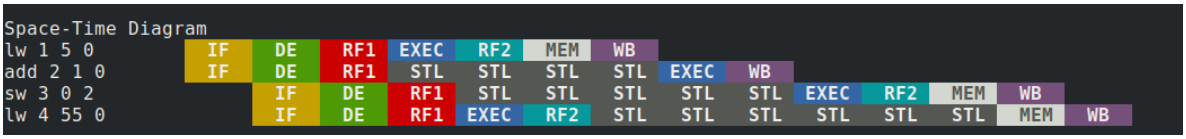

Figure 5: Uncomputed target address of `sw` stalls all future memory instructions

In figure 5, we see that a store instruction with an uncomputed memory address stalls all further load-store operations:

1. Here we see that the the `sw` instruction (third) is stalled as register 2 is busy due to former `add` instruction.
2. The important thing to realize here is that since at this point the address to be used by `sw` is non deterministic, we can't handle future load/store instructions because they might also use the same memory address and hence use the wrong value.
3. Therefore the `lw` instruction in this example has to stall until the memory address of `sw` has not been calculated and only then we check if it is possible to be sent to `MEM` unit or not.

Ex 4: Demonstrating the scheme for load and store operations (2)

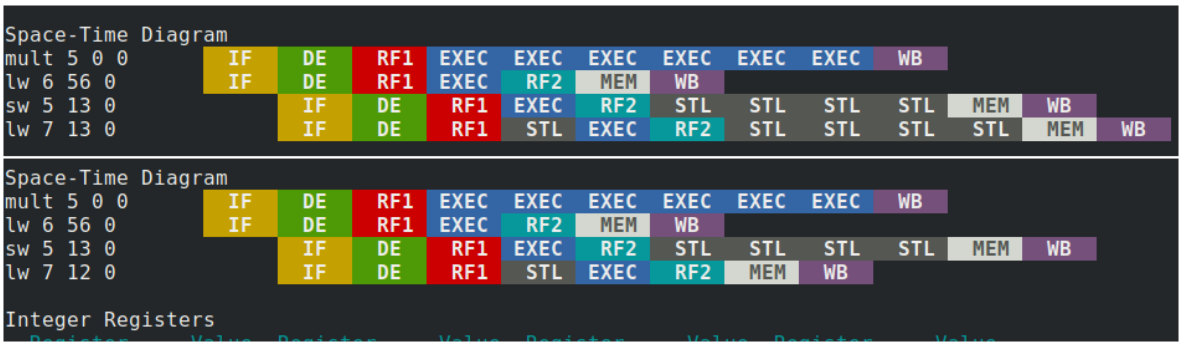

Figure 6: Example 4

In figure 6, we can see a store instruction with uncomputed source register (but computed memory address) only blocks instructions which have the same target memory address:

1. The `mult` instruction's destination register is 5 which is the source register in `sw` instruction. So `sw` instruction has to wait till `mult` graduates.
2. On the other hand, the memory address to which `sw` writes can be calculated irrespective of the `mult` instruction.
3. So now effectively there are two cases, either the `sw`, `lw` instructions use the same memory address (the former case) or they use different memory addresses (the latter case). If `lw` and `sw` have same address then the latter `lw` has to be stalled until the `sw` instruction has been completed. Otherwise we can send the `lw` instruction to memory unit which can be seen from the output.

Ex 5: Demonstrating handling of WAR hazards

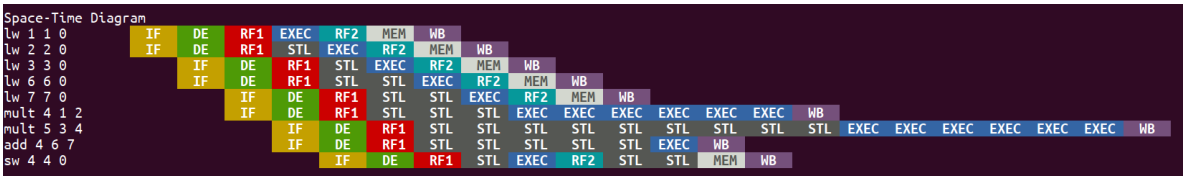

Figure 7: WAR hazards are easily solved by register renaming and remapping

In figure 7, we can see that instruction 8 has a WAR dependency on instruction 7 i.e. while instruction 7 waits for register 4 so that it can be read from, instruction 8 tries to write the result of addition to register 4.

1. Notice that instruction 8 doesn't wait for instruction 7 to complete — it writes to a different *physical* register instead. The logical-to-physical mapping is updated and used for future instructions.

2. Notice an extra stall after `RF1` stage in instruction 9 (as well as in earlier `lw` instructions). This arises due to inavailability of `ALU3` (which is used for calculating target addresses for memory instructions).

7 Results

Shown below is a sample run of SPSIM for a program having **nested `for`** loops along with other memory/integer operations. The space-time diagram and final states of registers and memory are also shown (see figure 8).

7.1 Input program

```
data:
1  1
2  3

main:
lw  1  1  0
lw  2  1  0
lw  5  2  0
lw  3  0  0
lw  4  1  0
lw  1  1  0
add  3  3  4
add  1  1  4
bne  1  5  -3
add  2  2  4
bne  2  5  -6
sw  3  6  0
```

7.2 Output

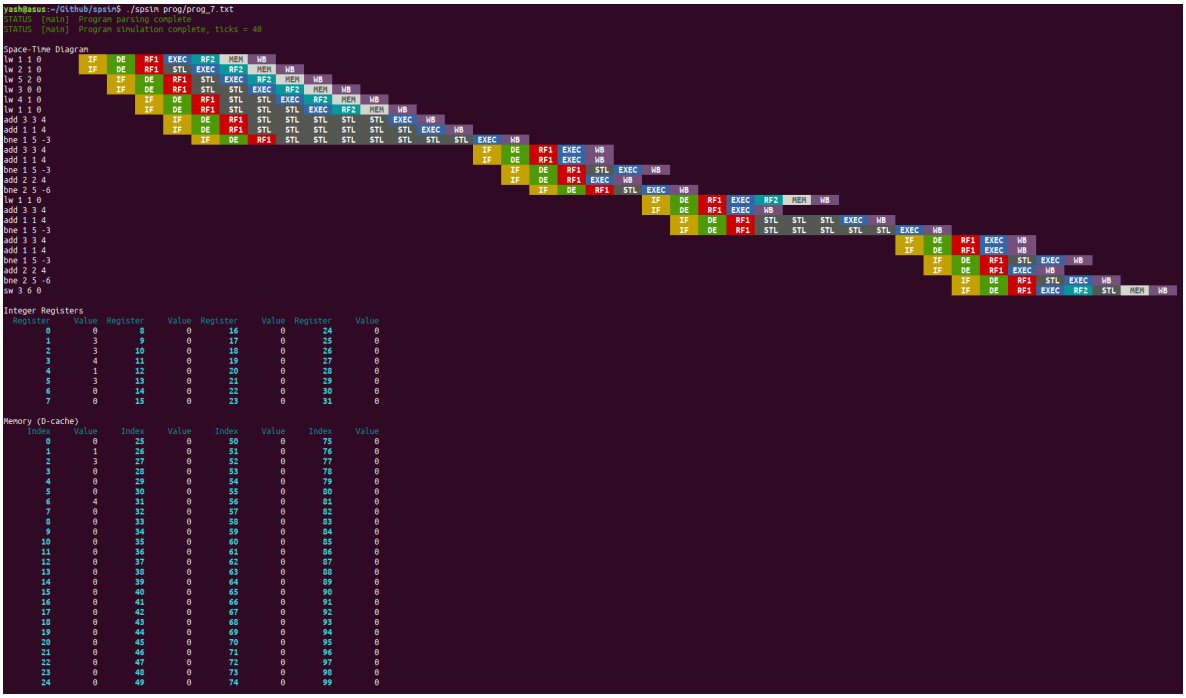

Figure 8: Output of sample run

8 Contributions

All group members have thorough knowledge on the working of the simulator. All polices were discussed in the group before being implemented. Contributions mentioned below only highlight the work put by members in coding up things. We have maintained a GitHub repository for this project (`https://github.com/ys1998/spsim`). For further details on each member's work, you may look at the `Contributors` tab or the commit history. Note that contributions to a particular topic also include the painful testing process for making the code bug-free.

Yash Shah, *160050002*

- Modularized the code and introduced *clocked* and *static* entities and pointer-based *connections* based on the initial draft prepared by Utkarsh.
- Collaborated with Rupesh to introduce branch prediction and branch instructions into the ISA.
- Worked on state restoration and flushing of invalid instructions in case of incorrect speculation for integer and `lw` operations.
- Handled the output and pretty-printing of space-time diagram and the final state of registers and memory.

Naman Jain, *160050025*

- Worked on adding the load and store instructions to pipeline

Utkarsh Gupta, *160050032*

- Implemented the first working superscalar simulator from scratch which had a fully pipelined execution model for unicycle integer instructions.
- The implementation didn't have branch prediction and support for load/store instruction; that was augmented later on by other members.
- Setup the most basic integer ISA, input output format, and the pipelining structure so that the code was easily extendable.

Rupesh, *160050042*

- Collaborated with Yash to introduce branch prediction and branch instructions into the ISA.
- Worked on state restoration and flushing of invalid instructions in case of incorrect speculation for integer, `lw` and `sw` operations.

Sharvik Mital, *160050059*

- Worked on adding the load and store instructions to pipeline

References

[1] Kenneth C. Yeager. The mips R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–41, 1996.

[2] Milos Pruvolic. Tomasulo algorithm. *Lecture Notes*.

[3] Milos Pruvolic and Catherine Gamboa. High performance computer architecture. *Video Lectures*.

[4] Onur Mutlu. Computer architecture. *Video Lectures*.

[5] Susan Eggers. Out-of-order execution & tomasulo algorithm. *Lecture Notes*.

[6] Nima Honarmand. Memory accesses in out of order execution. *Lecture Notes*.