CS G524 Advanced Computer Architecture

Lab Sheet 2

Semester 2 – 2019-20

Goal: Understand Instruction Pipelining using DrMIPS tool

Background:

QtSPIM does not provide different alternatives for experimenting with Hazards, Pipelines etc. Instead we will use another emulation tool DrMIPS.

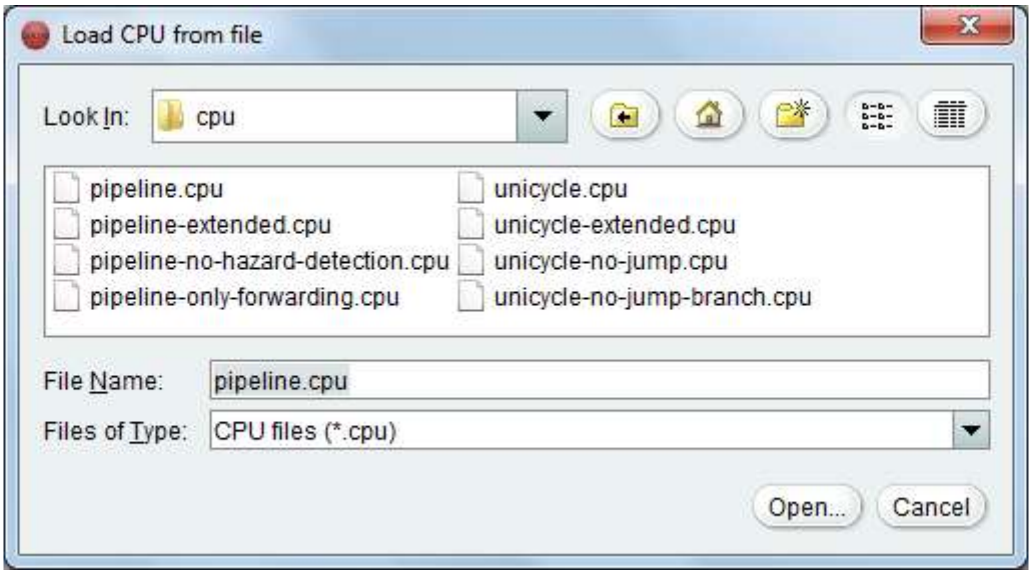
Installation:

1. Download DrMIPS files from https://pkgs.org/download/drmips
2. Setup drmips_2.0.1-2_all.deb
3. Launch the tool using: drmips

NOTE: When compared to QtSPIM, DrMIPS skips most of the instructions and all the system calls. Hence older code may need to be tweaked before running. It is primarily to explore the impact of pipelines, hazards etc.

Use Shift+f1 to check the supported instructions, pseudo instructions and directives.

Some Key features:



pipeline.cpu
pipeline-extended.cpu
pipeline-no-hazard-detection.cpu
pipeline-only-forwarding.cpu
unicycle.cpu
unicycle-extended.cpu
unicycle-no-jump.cpu
unicycle-no-jump-branch.cpu

File Name: pipeline.cpu
Files of Type: CPU files (*.cpu)

The datapaths provided by default are: (CPU->Load)

- Unicycle datapaths:
  - unicycle.cpu: the default unicycle datapath.
  - unicycle-no-jump.cpu: simpler variant of the unicycle datapath that doesn't support the j (jump) instruction.
  - unicycle-no-jump-branch.cpu: an even simpler variant of the unicycle datapath that doesn't support jumps nor branches.

unicycle-extended.cpu: a variant of the unicycle datapath that supports some additional instructions, like multiplications and divisions.

Pipeline datapaths:

pipeline.cpu: the default pipeline datapath, which implements hazard detection. The pipeline datapaths don't support the j (jump) instruction.

pipeline-only-forwarding.cpu: variant of the pipeline datapath that, in terms of hazard detection, only implements data forwarding (giving wrong results).

pipeline-no-hazard-detection.cpu: a variant of the pipeline datapath that doesn't implement any form of hazard detection at all (giving wrong results).

pipeline-extended.cpu: a variant that supports some additional instructions, like unicycle-extended.cpu.

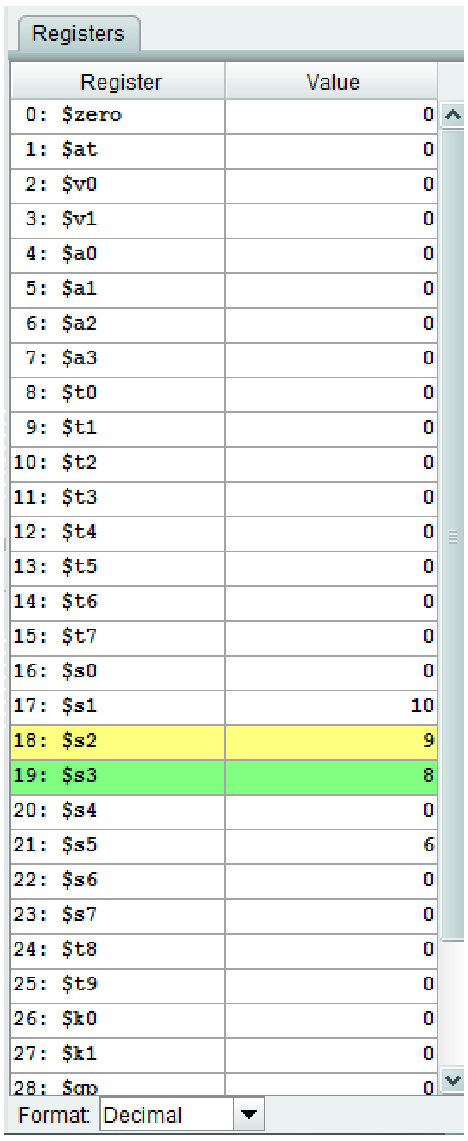
Registers and Data Memory:

The values that are currently being accessed are highlighted in different colors. The colors mean:

Green: the register/address is being read by the register bank/data memory.

Red: the register/address is being written to the register bank/data memory.

Yellow: the register/address is being read and written in the same cycle in the register bank/data memory.


Registers

Register	Value
0: $zero	0
1: $at	0
2: $v0	0
3: $v1	0
4: $a0	0
5: $a1	0
6: $a2	0
7: $a3	0
8: $t0	0
9: $t1	0
10: $t2	0
11: $t3	0
12: $t4	0
13: $t5	0
14: $t6	0
15: $t7	0
16: $s0	0
17: $s1	10
18: $s2	9
19: $s3	8
20: $s4	0
21: $s5	6
22: $s6	0
23: $s7	0
24: $t8	0
25: $t9	0
26: $k0	0
27: $k1	0
28: $gp	0

Format: Decimal

Pipeline color coding:

If simulating a pipeline processor, all the instructions that are in the pipeline are highlighted in different colors, each one representing a different stage. The different colors mean:

Blue: Instruction Fetch (IF) stage.

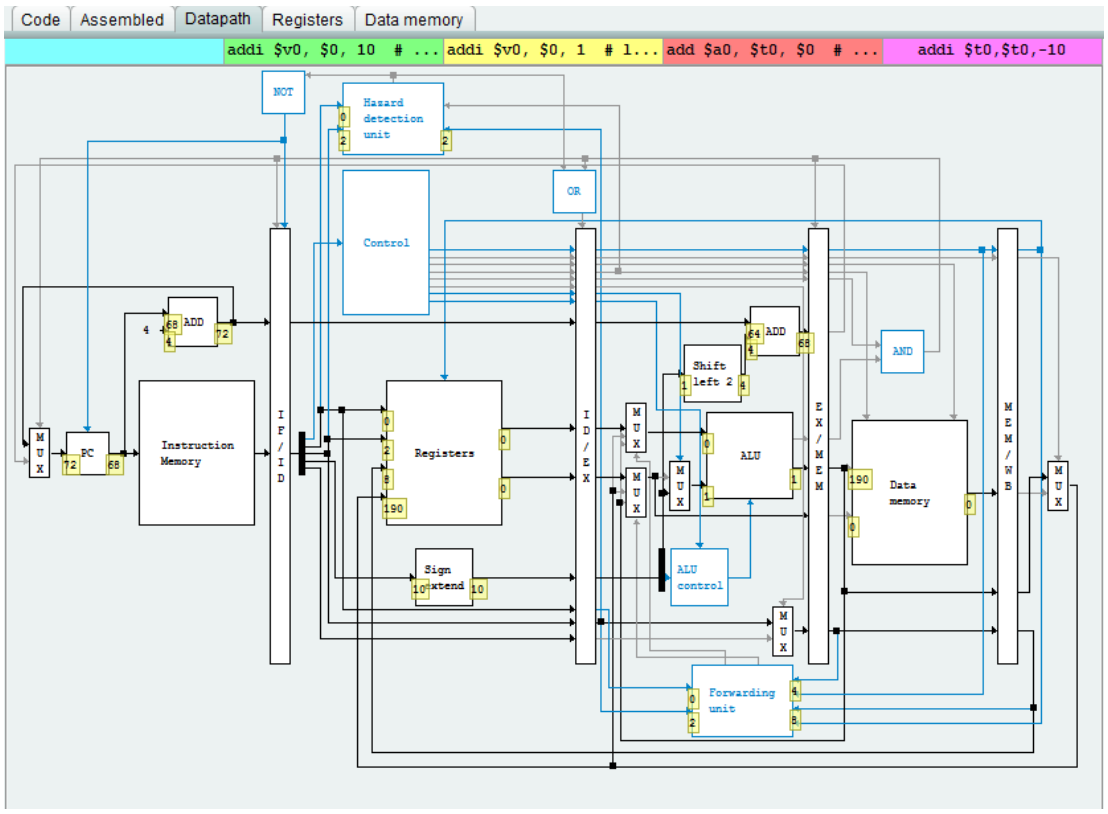
Green: Instruction Decode (ID) stage.

Yellow: Execute (EX) stage.

Red: Memory access (MEM) stage.

Magenta: Write Back (WB) stage.

Ref: https://github.com/brunonova/drmips and the documentation with tool itself.



<u>Exercise 1</u>: Preload the following registers with given values. (After assemble, double click on the register to load it with following values, then step-wise run the code)

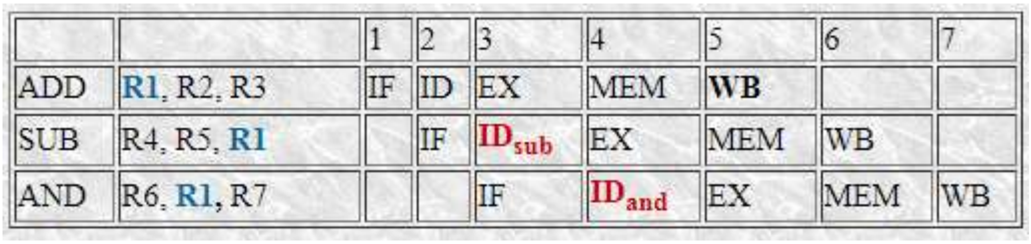
$s1 => 10	$s2 => 9
$s3 => 8	$s4=>63
$s5 => 6	

```
.data

.text

main:

add $s1,$s2,$s3

sub $s7,$s5,$s1

and $s6,$s1,$s4

li $v0,10
```

Execute the above piece of code:

- With CPU as pipeline-no-hazard-detection.cpu
- With CPU as pipeline-only-forwarding.cpu

Notice the change in values of $s1,$s7,$s6 in both cases, general datapath(without stalls) for above instructions are:



		1	2	3	4	5	6	7
ADD	R1, R2, R3	IF	ID	EX	MEM	WB		
SUB	R4, R5, R1		IF	ID_{sub}	EX	MEM	WB	
AND	R6, R1, R7			IF	ID_{and}	EX	MEM	WB

Exercise 2: Install the tool and run simple code.
The code should load two words from memory (.data .word directives) and add them.
If the sum is more than 100 (bge) reduce the first value by 10 and store back in memory.
Load the code, Assemble it (Execute-> Assemble) and then step through it. Observe how registers and data paths get impacted.

```
.data
   n0: .word 100
   n1: .word 200
   n2: .word 100
.text
   main:
      lw $t0,n0
      add $t0,$t0,$t0 # forwarding hazard!!, if only-data-forwarding CPU is used
      lw $t1,n1
      lw $t2,n2
      add $a0,$t0,$t1
      bge $a0,$t2,reduce
      add $t1,$t1,$t1 #dummy instruction should be flushed on correct CPUs
      ble $a0,$t2,exit
   reduce:
      addi $t0,$t0,-10
```

Reference:
http://web.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/forward.html

Other Problems as take home – if you love challenges (non-evaluative):
Exceptions revisited
Reference: http://www.cs.iit.edu/~virgil/cs470/Labs/Lab7.pdf
1. Write your code to declare a variable called MAX_INT with the initial value the maximum value of a signed integer. Loads the value of MAX_INT in register $t0 and adds it to itself (use signed addition). Observe whether you got Trap or an Interrupt.
2. Similarly attempt small code to create other exceptions - you can use Lab week 7 as reference.

Understanding memory mapped IO
Reference: http://www.cs.iit.edu/~virgil/cs470/Labs/Lab7.pdf
Do the memory mapped IO as given in page 10, Step 6 of reference material (Laboratory - PreLab) from Virgil Bistriceanu.
Do the Laboratory 7: Inlab module from above reference for "Interrupt driven output"