Carl Chan (c73chan)
Tony Tao (l5tao)

Contents

Demo

The demo was held on Friday July 25, 2014 at 3:40 pm.

Note: The implementation worked correctly in the demo. It did, however, use forwarding through the register file. We have kept it like this for the submission as the request of the professor.

To run the simulation, simply double-click "modelsim.bat". To set which benchmark to run, set the SREC file on line 54 of "ECE429_CPU_tb.v".

Source Code

The source code is included in the sub-folders. All of them are Verilog files with names prefixed with "ECE429_", with the following suffixes:

Suffix	Contents
ALU	ALU Module
ControlBits	The definition of the bit vector used for control in the CPU
Decode	Decode module
Fetch	The fetch module
Memory	Memory module (two instances used in CPU)
RegFile	Register file module
SREC_Parser	Converts SREC file into form loadable to memory
CPU	The CPU module which uses the above components
CPU_tb	Top-level test bench of CPU.

Testing

Register File: Initial

```
# r          0:          0
# r          1:          1
# r          2:          2
# r          3:          3
# r          4:          4
# r          5:          5
# r          6:          6
# r          7:          7
# r          8:          8
# r          9:          9
# r         10:         10
# r         11:         11
# r         12:         12
# r         13:         13
# r         14:         14
# r         15:         15
# r         16:         16
# r         17:         17
# r         18:         18
# r         19:         19
# r         20:         20
```

```
# r        21:         21
# r        22:         22
# r        23:         23
# r        24:         24
# r        25:         25
# r        26:         26
# r        27:         27
# r        28:         28
# r        29: 2148663296
# r        30:         30
# r        31:          0
```

Comment [C1]: (0x80020000 + 1MB)

Comment [C2]: Initialized to 0 at start so we can tell when the overall program is done

Register File: After fact

```
# r         0:          0
# r         1:          1
# r         2:          0
# r         3:      40320
# r         4:          0
# r         5:          5
# r         6:          6
# r         7:          7
# r         8:          8
# r         9:          9
# r        10:         10
# r        11:         11
# r        12:         12
# r        13:         13
# r        14:         14
# r        15:         15
# r        16:         16
# r        17:         17
# r        18:         18
# r        19:         19
# r        20:         20
# r        21:         21
# r        22:         22
# r        23:         23
# r        24:         24
# r        25:         25
# r        26:         26
# r        27:         27
# r        28:         28
# r        29: 2148663296
# r        30:         30
# r        31:          0
```

Comment [C3]: $v0 is cleared before exiting program

Comment [C4]: This holds (n-1)!, where n corresponds to last factorial calculated. The last factorial calculated is 9!, which is correct as (9-1)!=40320.

Comment [C5]: Stack pointer restored to its old value of (0x80020000 + 1MB)

Comment [C6]: $ra returned to its old value (initialized to 0 at start so we can tell when the overall program is done).

Register File: After Simple If

```
# r         0:          0
# r         1:          1
# r         2:          7
# r         3:          5
# r         4:          4
# r         5:          5
# r         6:          6
# r         7:          7
# r         8:          8
```

```
# r          9:          9
# r         10:         10
# r         11:         11
# r         12:         12
# r         13:         13
# r         14:         14
# r         15:         15
# r         16:         16
# r         17:         17
# r         18:         18
# r         19:         19
# r         20:         20
# r         21:         21
# r         22:         22
# r         23:         23
# r         24:         24
# r         25:         25
# r         26:         26
# r         27:         27
# r         28:         28
# r         29: 2148663296
# r         30:         30
# r         31:          0
```

Register File: After Simple Add

```
# r          0:          0
# r          1:          1
# r          2:          5
# r          3:          3
# r          4:          4
# r          5:          5
# r          6:          6
# r          7:          7
# r          8:          8
# r          9:          9
# r         10:         10
# r         11:         11
# r         12:         12
# r         13:         13
# r         14:         14
# r         15:         15
# r         16:         16
# r         17:         17
# r         18:         18
# r         19:         19
# r         20:         20
# r         21:         21
# r         22:         22
# r         23:         23
# r         24:         24
# r         25:         25
# r         26:         26
# r         27:         27
# r         28:         28
# r         29: 2148663296
# r         30:         30
```

```
# r         31:          0
```

Register File: After Bubble Sort

```
# r          0:          0
# r          1:          1
# r          2:          0
# r          3:          8
# r          4: 2148663256
# r          5:          8
# r          6:          6
# r          7:          7
# r          8:          8
# r          9:          9
# r         10:         10
# r         11:         11
# r         12:         12
# r         13:         13
# r         14:         14
# r         15:         15
# r         16:         16
# r         17:         17
# r         18:         18
# r         19:         19
# r         20:         20
# r         21:         21
# r         22:         22
# r         23:         23
# r         24:         24
# r         25:         25
# r         26:         26
# r         27:         27
# r         28:         28
# r         29: 2148663296
# r         30:         30
# r         31:          0
```

Register File: After Check Vowel

```
# r          0:          0
# r          1:          1
# r          2:          0
# r          3:          0
# r          4: 1800826743
# r          5: 1130915171
# r          6:          6
# r          7:          7
# r          8:          8
# r          9:          9
# r         10:         10
# r         11:         11
# r         12:         12
# r         13:         13
# r         14:         14
# r         15:         15
# r         16:         16
# r         17:         17
# r         18:         18
```

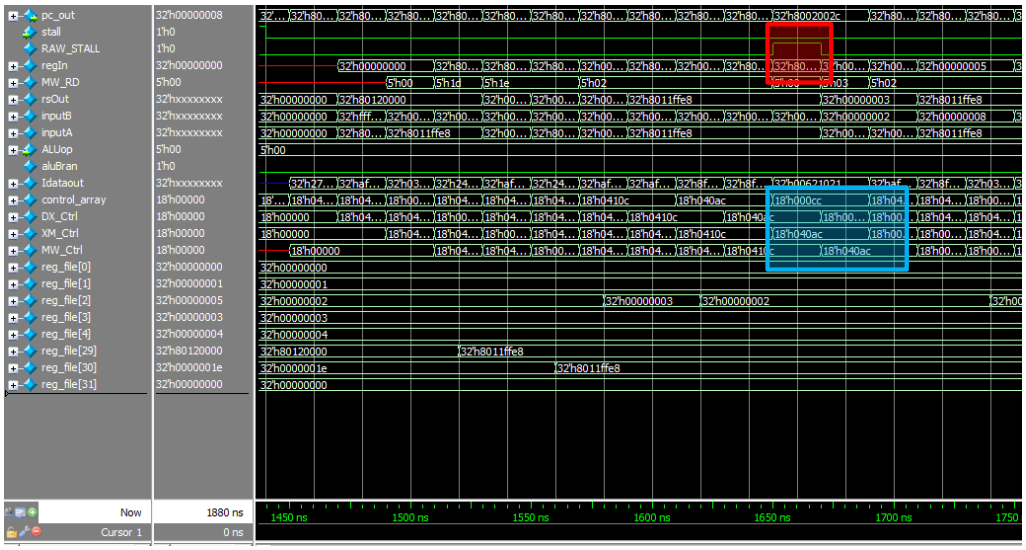
```
# r         19:          19
# r         20:          20
# r         21:          21
# r         22:          22
# r         23:          23
# r         24:          24
# r         25:          25
# r         26:          26
# r         27:          27
# r         28:          28
# r         29: 2148663296
# r         30:          30
# r         31:           0
```

Example of Stall in Fully Bypassed CPU (Simple Add)

This is a load-use stall in the full-bypassed CPU. The red square highlights the assertion of the stall signal for a load-use stall ("RAW_STALL"). The instructions are $lw\ v0{,}4(s8)$ followed by $addu\ v0, v1, v0$.

The blue rectangle shows the control array from the decoder ("control_array") remaining the same while the load-use stall is in effect, while the control arrays of the later stages are allowed to move in. The value control array clocked into the execute stage while the stall is active corresponds to a NOP.

There were no stalls for RAW dependency between instructions in the write-back and decode stages. This is as we implemented forwarding through the register file. This was discussed with the professor during the demonstration.



Example of Forwarding (Simple Add)

The assertion of the signal in the red rectangle causes an M/X bypass for the instruction $addiu\ sp, sp, -24$, followed by $sw\ s8{,}20(sp)$. In this case, the output of the first instruction is

forwarded to the Rs input of the second instruction to avoid a stall. Notice how the stall signal "RAW_STALL" is not asserted, unlike in the previous example.

The blue rectangle shows the assertion of the signal which enables a W/X bypass. This is for the RAW dependency between $addiu\ sp, sp, -24$ and the instruction after it, $move\ s8, sp$. Notice again how the "RAW_STALL" is not asserted, showing that the bypass succeeded without the need to stall.

The orange rectangle shows the assertion of the signal for a W/M bypass. This is for the instructions $li\ v0,3$ followed by $sw\ v0,0(s8)$, where the latter must take in the result of the prior to know what value to load into memory.