CS422: Computer Architecture

Homework 3

Implementation Details - Part B

We implement 5 designs:

1. Design 1: Full interlock design, two delay slots for branches (one stall, one filled by compiler)
2. Design 2: Full interlock design, one delay slot for branches
3. Design 3: Design 2 with EX-EX Bypass
4. Design 4: Design 3 with MEM-EX Bypass
5. Design 5: Design 4 with MEM-MEM Bypass

The heart of the pipelined design is `pipe_reg_t`, a data structure describing the pipeline registers, defined in mips.h. There are four pipeline registers inside `Mipc`, namely `_fd`, `_de`, `_em` and `_mw`. Although in a practical design each of these pipeline registers would forward different data fields between stages, implementing them as different data structures would involve a lot of code duplication (such as copy constructors), hence we have used only a single structure `pipe_reg_t` for all, essentially a superset.

All pipeline registers are written to in the negative half of the clock cycle. This makes it safe for all stages to read from their pipeline registers during the positive halfs. Each stage reads/writes from its two (one) pipeline regs, similar to a real design.

All system calls are deferred till the negative half of WB. This means each syscall takes up 5 cycles, without sharing its cycles with any other instruction.

Updating the program counter

IF works in the negative half, whereas EX does it works in the positive half. This allows for branch targets to be updated without an additional stall. Additionally, instead of incrementing the PC by 4 in IF, this is also done by EX to keep the design clean & centralized.

Hazard detection & interlock

Hazard detection is done in DE. For this, we use `_gpr_wait[32]` (and similarly for non-gprs); `_gpr_wait[i]` is a counter which holds how many cycles to stall before it is safe to read gpr `i` from the register file. As WB writes to the register file in the positive half but decode works in the negative half, data hazards can lead to a maximum stall of 2 cycles.

Hazards are detected in DE if `_gpr_wait[...]` is non-zero for any of the source operands. `_gpr_wait` is decremented every cycle, and the entry corresponding to the current instruction's destination is also updated in DE. Hazards (if any) are detected in the +ve half of DE - this allows us to stall the pipeline before the next instruction is even fetched. This also means that for syscalls there is no instruction to be flushed.

Bypasses

We implement three bypasses EX-EX, MEM-EX and MEM-MEM - although the names are slight misnomers? For EX-EX, we bypass the output operands calculated in the positive half of EX to DE during its negative half. This is essentially from the output of EX to the input of EX, although technically the "receive" part of the bypass is done in DE.

Similarly, MEM-EX bypasses to DE, whereas MEM-MEM bypasses to EX.

All detection is done during DE, including for MEM-MEM bypasses. The detection is done very simply using `_gpr_wait` - for example, if the current instruction has a source operand `i`, and `_gpr_wait[i]` is 2, then we know that the instruction currently in EX produces gpr `i` and we trigger the EX-EX bypass. Similarly for MEM-EX.

MEM-MEM bypass is triggered when an EX-EX bypass is triggered, but when the instruction in EX is a load and the instruction in DE is a store (with a few more corner cases).

All bypasses are implemented for gprs, fprs, `_hi` and `_lo`. Interlock logic for load delay hazards exists, and would cause a stall of one cycle if detected.

Results Explanation - Part B

CPI gradually reduced till it becomes ~1 as expected. Each syscall uses up a total of 5 cycles, which causes the CPI to be greater than 1, particularly in programs with a lot of system calls.

One thing to note is that the MEM-MEM bypass is never triggered (and also the load delay interlock) for the 14 given programs. This is because it seems like the compiler is ensuring that the instruction in the load-delay slot is independent of the load.

We tried to trigger the MEM-MEM bypass by adding a load followed by a store in example.S in asm-sim - however the compiler added a nop after the load, confirming our hypothesis. Nevertheless, we have left our MEM-MEM bypass logic as the assignment states that we are not to assume that the instruction is independent.

The load interlock is triggered only by Subreg - this is also due to an unusual sequence of instructions:

```
401074:    88a20000    lwl $v0,0($a1)
401078:    98a20003    lwr $v0,3($a1)
```

As `$v0` is both a source and destination for both `lwl`, `lwr`, this triggers the load interlock.

Part A Results

All the fractions have been listed as percentages.

Test Case	Total No. of Instructions	Loads (%)	Stores (%)	Conditional Branches (%)
asm-sim	158	39.24050633%	34.81012658%	0%
c-sim	1928	17.42738589%	8.19502075%	19.29460581%
endian	1965	19.13486005%	11.043257%	18.01526717%
factorial	2530	11.10671937%	8.61660079%	18.41897233%
fib	44903	13.59151949%	17.77609514%	8.34465403%
hello	1744	19.094037%	8.60091743%	19.61009174%
host	14894	20.10205452%	9.92345911%	19.22250571%
ifactorial	2420	16.48760331%	7.85123967%	19.25619835%
ifib	7581	16.98984303%	10.01187178%	17.50428703%
log2	2096	17.89122137%	8.34923664%	19.60877863%
msort	17219	10.58714211%	5.44166328%	18.12532667%
rfib	25668	14.39925199%	17.02898551%	9.51379149%
subreg	4361	21.64641137%	16.28066957%	15.59275395%

Test Case	Total No. of Instructions	Loads (%)	Stores (%)	Conditional Branches (%)
towers	82763	20.05727197%	8.48809250%	20.54299327%
vadd	7259	6.14409698%	16.35211462%	10.57996969%

Part B Results

CPI of all the design stages is given in the table below:

Test Case	Design-1	Design-2	Design-3	Design-4	Final Design	Load Delay Stalls
asm-sim	1.57	1.47	1.30	1.22	1.22	0
c-sim	1.85	1.65	1.14	1.01	1.01	0
endian	1.78	1.58	1.14	1.01	1.01	0
factorial	1.82	1.63	1.14	1.01	1.01	0
fib	1.52	1.41	1.18	1.00	1.00	0
hello	1.81	1.62	1.15	1.01	1.01	0
host	1.60	1.45	1.15	1.01	1.01	0
ifactorial	1.83	1.65	1.14	1.01	1.01	0
ifib	1.71	1.54	1.14	1.00	1.00	0
log2	1.78	1.59	1.14	1.01	1.01	0
msort	1.92	1.72	1.05	1.00	1.00	0
rfib	1.54	1.43	1.18	1.00	1.00	0
subreg	1.58	1.43	1.14	1.01	1.01	3
towers	1.55	1.41	1.15	1.00	1.00	0
vadd	1.98	1.88	1.15	1.00	1.00	0

While we assumed that the compiler did not insert independent instructions in the load delay slot, we noticed via some testing using handwritten assembly that it inserted NOPs in the load delay slot. This can be seen in the above table from the last column being all zeroes, except for Subreg. This effectively meant that MEM-MEM bypass or load interlock stalls would never be called. However, a close approximate to the number of load interlock stalls can be found using the number of NOPs immediately following a load instruction. This has been reported in the column `Load Interlock Stalls Estimate`.

Handwritten Assembly section:

```
.data
my_var: .word 0
        ...
```

```
main:
        la $7, my_var
        li $4, 1      # parameter passed to printf in $a0
        la $5, lab    # load the address of the string to be printed in $a1
        li $6, 9      # length of the string in $a2
        sw $6, 0($7)
        lw $7, 0($7)
        sw $7, 0($7)
        li $2, 1004   # load the system call number in $v0
        ...
```

Corresponding section of `<main>` from the disassembly

```
00401020 <main>:
  ...
  40103c:   8ce70000    lw  $a3,0($a3)
  401040:   00000000    nop
  401044:   ace70000    sw  $a3,0($a3)
  ...
```

Under the assumption that the compiler does not insert independent instructions in the load delay slot, the total number of instructions that have been executed by the processor can be given by `(total instructions - load interlock stalls)`, since each load interlock stall corresponds to 1 extra NOP.

Thus in the final column, we have included an effective CPI which is calculated using `(No. of simulated cycles) / (No. of instructions - No. of load interlock stalls)`, and as can be seen these are slightly higher than 1 due to the introduction of load interlock stalls.

This estimated count of (not observed) Load-Interlock stall cycles and syscalls with respect to total no. of instructions in the final design are given below:

Test Case	No. of Instructions	Syscalls(%)	Load Interlock Stalls Estimate (%)	Effective CPI
asm-sim	50	6.00%	0.00%	1.22
c-sim	1826	0.27383%	7.77656%	1.10
endian	1865	0.2681%	7.61394%	1.09
factorial	2408	0.20764%	6.60299%	1.08
fib	44780	0.01117%	0.98928%	1.01
hello	1638	0.30525%	8.54701%	1.11
host	14520	0.16529%	5.97107%	1.07
ifactorial	2293	0.21805%	6.93415%	1.08
ifib	7453	0.06709%	5.94391%	1.07

Test Case	No. of Instructions	Syscalls(%)	Load Interlock Stalls Estimate (%)	Effective CPI
log2	1996	0.2505%	7.86573%	1.10
msort	17103	0.02923%	3.82974%	1.04
rfib	25541	0.01958%	1.73447%	1.02
subreg	4224	0.14204%	5.5161%	1.06
towers	80652	0.08307%	5.83866%	1.06
vadd	7130	0.07013%	2.48247%	1.03