CS422 Assignment 3

Pragati Agrawal (220779), Dhruv Gupta (220361)

26 March 2025

PART-A : Unpipelined

subreg.c

For writing a code which generates `lwl, lwr, swl, swr` instructions we first tried to do unaligned memory accesses using an integer pointer. However, this strategy did not work, since compiler always generates a `lw/sw` instruction for integer pointer access. We then looked at the library functions where these instrunctions were used and found that `memcpy` is one such function. When we use `memcpy` to copy a constant string to a char pointer, the compiler first uses `lwl, lwr, swl, swr` instructions to create the string and then instead of calling the library function, inserts the code into the main function itself, possibly due to some optimization. Hence the final code generated contains the `lwl, lwr, swl, swr` instructions in the main code.

Statistics

We can note the difference in number of instruction in `rfib` and `ifib` - `rfib` has more number of instructions due to recursion. Also the absolute number of loads and stores is also higher in `rfib`. The recursive code is therefore less efficient than `ifib` as expected (exponential vs quadratic time complexity). However, the difference between `factorial` and `ifactorial` is less pronounced, since the the time complexities of both codes is linear.

Program	Total Instr.	% Loads	% Stores	% Branches
asm-sim	182	40.66%	36.81%	0.00%
c-sim	1952	17.83%	8.71%	19.06%
factorial	2554	16.80%	9.01%	18.25%
hello	1768	19.51%	9.16%	19.34%
ifactorial	2444	16.82%	8.27%	19.07%
log2	2120	18.25%	8.82%	19.39%
rfib	25692	14.43%	17.06%	9.50%
towers	82787	20.07%	8.50%	20.54%
endian	1989	19.51%	11.51%	17.80%
fib	44927	13.61%	17.79%	8.34%
host	14918	20.15%	9.99%	19.19%
ifib	7605	17.09%	10.14%	17.45%
msort	17243	10.64%	5.50%	18.10%
subreg	2476	21.08%	9.57%	18.30%
vadd	7283	6.29%	16.46%	10.55%

PART-B : Pipelined

We implemented the pipeline incrementally. All the intermediate designs are listed below -

- **Design 1 :** Branch interlock enabled and interlock logic for stalls due to data hazards (including load-delay interlock)
- **Design 2 :** Branch interlock cycle removed from Design 1
- **Design 3 :** EX-EX bypass added to Design 2
- **Design 4 :** MEM-EX bypass added to Design 3
- **Design 5 :** MEM-MEM bypass added to Design 4.

Design 5 is our final design, which has no branch delay interlock, and also has all bypass paths enabled. Following table shows the CPI for all the designs and the percentage of load-delay interlock cycles for the final design -

Program	Design 1	Design 2	Design 3	Design 4	Design 5	% Load Stalls (D5)
asm-sim	1.50	1.40	1.28	1.16	1.16	0.00%
c-sim	1.80	1.65	1.14	1.01	1.01	0.00%
factorial	1.79	1.63	1.14	1.01	1.01	0.00%
hello	1.77	1.61	1.15	1.01	1.01	0.00%
ifactorial	1.80	1.64	1.14	1.01	1.01	0.00%
log2	1.74	1.59	1.15	1.01	1.01	0.00%
rfib	1.63	1.43	1.18	1.00	1.00	0.00%
towers	1.59	1.41	1.15	1.00	1.00	0.00%
endian	1.73	1.58	1.14	1.01	1.01	0.00%
fib	1.63	1.41	1.19	1.00	1.00	0.00%
host	1.62	1.45	1.15	1.01	1.00	0.00%
ifib	1.68	1.54	1.15	1.00	1.00	0.00%
msort	1.86	1.72	1.12	1.00	1.00	0.00%
subreg	1.71	1.54	1.16	1.01	1.01	0.00%
vadd	1.97	1.88	1.23	1.00	1.00	0.00%

Details of implementation

We have made following changes to get the final pipelined design -

- **Pipeline Registers :** We have made a class `PipeReg` to implement the pipeline registers. Each pipeline register keeps all the variables used by any of the stages. This makes it easier for the values to flow through the pipeline and each stage simply uses the variables it needs.

We have 8 such pipeline objects - 2 each for IF/ID, ID/EX, EX/MEM, MEM/WB, one from which the pipeline stage reads and another to which it writes. For eg., at the posedge, decoder takes the values it needs from IF/ID-w register into the IF/ID-r register, which it uses. Then at the nededge, it copies the contents to IF/ID-r to ID/EX-w register which is then used by the executor in the next cycle for executing this instruction. Other pipeline stages also work similarly. This implementation avoids any race throughs.

- **Branch Interlock :** A flag `_lastbdslot` is maintained which is set to one if the last instruction was branch delay slot instruction. It is set by the executor in the positive cycle and checked by the fetcher in the negative cycle. If fetcher finds it set to 1, it stalls.

- **Data Hazards, Load-Delay interlock and Bypasses :** Decoder, in the positive half cycle decodes the instruction to get the source and destination registers. For this some extra logic in decode function was added to extract the register numbers. Decoder contains a function `detectStall()`, which compares the source registers of the current instruction with the destination of the previous two instructions and checks for data hazards. If the corresponding value cannot be taken from a bypass or the read register file it tells us to stall, otherwise it is noted that this value should be taken from the bypass or the read register file. For this it maintains a bypass variable for each possible source, which tracks which bypass to take value from. The decoder sets a flag `_data_stall` to one if stall is needed. This flag is checked by the fetcher in the negative half and it stalls if it finds the flag set to 1.
  If a bypass is needed, the decoder sets the correct source stage from which the bypassed value should be taken by that instruction in a `_bypass` variable for each such source. The precedence order is that the `MEM-MEM` bypass contains the latest value, followed by the `EX-EX` bypass, and then comes the `MEM-EX` bypass.
  In the negative cycle the decoder once again calls the decode function, so that this time it also reads the correct values from the register files which would have been written by the Writeback stage in the positive cycle. Note that, even in the final design, the load-delay interlock cycle is only needed if the output of the MEM stage of previous instruction is needed by the EX stage of the current instruction.

- **Taking Bypass values :** The executor calls a function `pick_bypass_value()` which checks for each source if its bypass variable is set to EX-EX or MEM-EX, if yes, it takes the value from the corresponding pipeline register and overwrites the current value, otherwise it has to take the value filled in by the decoder, so it does nothing. The MEM stage also needs to take values from the MEM-MEM bypass. For this in each of the memOp functions, the value of the corresponding bypass variable is checked and the value is taken from appropriate pipeline register if it is set to MEM-MEM.

- **Syscalls :** The fetcher needs to be stalled if a syscall is present in the pipeline. For this a flag `_syscall_present` is set by the decoder if it finds a syscall. It is reset by the Writeback stage.

- **Flushing the pipeline registers :** If a stall is detected (due to any reason) by the fetcher, it flushes the IF/ID-w pipeline register which has an effect of sending a no-op through the pipeline. Similarly if a data stall is detected by the decoder it flushes the ID/EX-w pipeline register. For flushing a pipeline register, the `PipeReg` class has a method `flush_regs()` which resets all the variables in the pipeline register to their default values.

We have defined macros in `mips.h` which enable us to switch across designs-

- `BRANCH_INTERLOCK_ENABLED` : If set to 0, the branch interlock cycle is removed.
- `BYPASS_EX_EX_ENABLED` : If set to 0, the EX-EX bypass is removed.
- `BYPASS_MEM_EX_ENABLED` : If set to 0, the MEM-EX bypass is removed.
- `BYPASS_MEM_MEM_ENABLED` : If set to 0, the MEM-MEM bypass is removed.

Comparison of All Bypass Path Combinations

The way we have implemented the bypasses and the branch interlock, makes it possible to enable or disable any of the bypass or the branch interlock independent of each other. Note that stalls due to data hazard (including the load-delay interlock cycles) are needed in all the designs for correctness. We have collected data for all 8 possible combinations of bypasses (with branch interlock disabled).

- **Design 1 :** Branch interlock cycle but no bypass path enabled
- **Design 2 :** EX-EX bypass added to Design 1
- **Design 3 :** MEM-EX bypass added to Design 1
- **Design 4 :** MEM-MEM bypass added to Design 1
- **Design 5 :** EX-EX + MEM-EX bypass added to Design 1
- **Design 6 :** MEM-EX + MEM-MEM bypass added to Design 1
- **Design 7 :** EX-EX + MEM-MEM bypass added to Design 1
- **Design 8 :** All bypasses enabled

Program	Design 1	Design 2	Design 3	Design 4	Design 5	Design 6	Design 7	Design 8
asm-sim	1.40	1.28	1.24	1.40	1.16	1.24	1.28	1.16
c-sim	1.65	1.14	1.27	1.64	1.01	1.27	1.14	1.01
factorial	1.63	1.14	1.26	1.63	1.01	1.26	1.14	1.01
hello	1.61	1.15	1.25	1.61	1.01	1.25	1.15	1.01
ifactorial	1.64	1.14	1.27	1.64	1.01	1.27	1.14	1.01
log2	1.59	1.15	1.24	1.59	1.01	1.24	1.14	1.01
rfib	1.43	1.18	1.15	1.43	1.00	1.15	1.18	1.00
towers	1.41	1.15	1.14	1.41	1.00	1.14	1.15	1.00
endian	1.58	1.14	1.24	1.58	1.01	1.24	1.14	1.01
fib	1.41	1.19	1.14	1.41	1.00	1.14	1.19	1.00
host	1.45	1.15	1.16	1.45	1.01	1.16	1.15	1.00
ifib	1.54	1.15	1.21	1.54	1.00	1.21	1.15	1.00
msort	1.72	1.12	1.33	1.72	1.00	1.33	1.12	1.00
subreg	1.54	1.16	1.21	1.54	1.01	1.21	1.16	1.01
vadd	1.88	1.23	1.41	1.88	1.00	1.41	1.23	1.00

Inferences

- None of the programs suffer from load-delay interlock stalls.
- However, for each program the CPI is slighly greater than 1. This is because of the stalls due to syscalls.
- We confirmed this by printing appropriate message (using the `MIPC_DEBUG` flag) to count the number of stalls due to syscalls. We observed that all the stalls were only due to syscalls. Moreover the number of instructions fetched + number of syscall stalls matches the number of simulated cycles (there is an extra instruction fetch at the end which we can ignore).
- For the programs in which CPI appears to be 1, it is due to low precision, because such programs have higher number of simulated instructions. They too have stalls due to syscalls, but this number does not scale to the number of instructions a program executes, so the `asm-sim` program, which executes just 58 instructions (8 stall cycles), sees a much high CPI than let's say `fib` which executes more than 44K instructions, but has overall 14 syscall stalls.
- MEM-MEM bypass is never used. We confirmed this similarly using debug prints.
- Only case when MEM-MEM bypass could be used is when a load is followed by a store and the store needs the value from the load. This would have an effect of copying a value from one memory location to another. Since MIPS has a large number of registers to keep values in, such an operation is rare and a smart compiler would optimize away such copy operations if possible.