CS422 Assignment 3

Abhishek Khandelwal (220040), Nipun Nohria(220717), Pallav Goyal(220747)

April 25, 2025

Implementation

To achieve the final pipelined design, we made several key modifications to the unpipelined version. Variables such as `decodedSRC1`, `decodedDST`, `branch_target`, and others used in the unpipelined design were converted into pipeline registers by creating arrays for each of them.

We carefully adhered to timing constraints between pipeline stages. Specifically, we ensured that functions in the `exec_helper` file for the EX and MEM phases do not read from and write to pipeline registers within the same half of a clock cycle. In our design, all reads from pipeline registers occur during the positive half of the cycle, while all writes to pipeline registers occur during the negative half. This is in contrast to the normal register file, which is written to in the positive half and read from in the negative half.

All potential data dependencies are identified during the positive half of the decode stage. We determine the type of bypass required and store this information in a `Bypass_information` struct, which also has its own pipeline register.

An important feature of our design is that no special handling is required for MEM-to-MEM bypasses. These bypasses typically occur in a load-store pair, where the store instruction reaches the MEM stage while the corresponding load reaches the WB stage. Since the WB stage executes in the positive half and the MEM stage in the negative half, the value is naturally available in time. Additionally, due to the way store instructions are implemented, we pass register numbers rather than values through the pipeline. Stores directly access the register file using `gpr[register_number]`, ensuring correct values are fetched without needing explicit bypassing.

Finally, our design makes it easy to observe the impact of a bypass versus a stall. To simulate a stall instead of performing a bypass, we simply need to uncomment a single line per bypass path. This simplifies debugging and experimentation.

1 Part I

Table 1: Statistics from part I. 3

Programs	Number of instructions	percentage of loads	percentage of stores	percentage of conditional branches
asm-sim	160	63(39.38%)	56(35.0%)	0(0%)
endian	1967	377(19.17%)	218(11.08%)	354(18.0%)
fib	44905	6104 (13.58%)	7983 (17.77%)	3747 (8.34%)
host	14896	2995 (20.11%)	1479 (9.93%)	2863 (19.21%)
ifib	7583	1289 (17.00%)	760 (10.02%)	1327 (17.50%)
msort	17221	1824 (10.59%)	938 (5.45%)	3121 (18.12%)
Subreg	1742	333 (19.10%)	158 (9.07%)	327 (18.77%)
vadd	7261	447 (6.15%)	1188 (16.36%)	768 (10.57%)
c-sim	1930	337 (17.46%)	159 (8.24%)	372 (19.27%)
factorial	2532	418 (16.51%)	219 (8.65%)	466 (18.41%)
hello	1746	334 (19.13%)	151 (8.65%)	342 (19.59%)
ifactorial	2422	400 (16.52%)	191 (7.89%)	466 (19.24%)
log2	2098	376 (17.93%)	176 (8.39%)	411 (19.59%)
rfib	25670	3697 (14.40%)	4372 (17.03%)	2442 (9.51%)
towers	82765	16601 (20.06%)	7026 (8.49%)	17002 (20.55%)

2 Part II

Given below is the table of CPI obtained at various iterations of our design:

d1 - The initial pipelined design with complete interlock logic. This has no bypass paths and has 2 delay slots for branches, one implemented as a stall cycle in the interlock.
d2 - The design obtained after removing the branch interlock cycle from d1.
d3 - The design obtained after adding MEM-EX bypass path in d2.
d4 - The design obtained after adding the EX-EX bypass path in d3.
d5 - The final design obtained after adding the MEM-MEM bypass path in d4.

Table 2: CPI statistics

Programs	d1	d2	d3	d4	d5
asm-sim	1.48	1.38	1.21	1.13	1.13
c-sim	1.79	1.64	1.27	1.01	1.01
endian	1.73	1.58	1.24	1.01	1.01
factorial	1.79	1.63	1.26	1.00	1.00
fib	1.63	1.41	1.21	1.00	1.00
hello	1.76	1.61	1.25	1.01	1.01
host	1.62	1.45	1.16	1.00	1.00
ifactorial	1.80	1.64	1.27	1.00	1.00
ifib	1.69	1.54	1.21	1.00	1.00
log2	1.74	1.58	1.23	1.01	1.01
msort	1.86	1.72	1.33	1.00	1.00
rfib	1.63	1.42	1.15	1.00	1.00
Subreg	1.77	1.61	1.25	1.01	1.01
towers	1.59	1.41	1.14	1.00	1.00
vadd	1.97	1.88	1.40	1.00	1.00

2.1 Observations

In our final design, there are still some programs where the CPI comes more than 1. Apart from load-delay interlock stalls, this is due to the the fact that during the detection of syscall and till it's execution, the pipeline gets filled with NOP instructions. Secondly, no instruction gets completed in the first 4 cycles of the program. So there will always be atleast 4 more cycles due to this reason and affect the CPI more for smaller programs like asm-sim. For example, a hypothetical program with 2 instruction will atleast have a CPI of 3.0 even without any stalls or NOPS.