Two Stage Pipelined Processor

Edison Lam
ELEC3608
500471825
University of Sydney

I. Introduction

The RISC-V architecture is an open, royalty-free Instruction Set Architecture (ISA) which was originally developed for education and research purposes [1]. This report discusses the implementation of the 32 bit version of RISC-V (RV32I), its operation and the implementation of a two stage instruction pipeline design into the RISC-V architecture as well as the implications of cache and suggests a suitable cache size for the processor. This report will also focus on resolving hazards caused by instruction pipelining using hardware interlocks and improving the performance of the pipeline process by incorporating the forwarding of results within this process. The results of each alteration to the design will also be discussed during this report.

This report uses a virtualised computer-aided design and SystemVerilog to simulate the operation different processors and to compare how implementing instruction pipelining into its design alters its operation as well as the optimal cache size for our processor. Specifically, this comparison will occur between the original Naive Educational RISC-V processor and our 2 stage pipelined processor with hardware interlock and forwarding functionalities based on its performance completing a given sub-process of a program. The outcome of the simulations displayed that the 2-stage pipeline processor completed the process slightly faster than the original single stage due to the 22.7% reduction in cycle period but had a lower CPI due to experience a significant number of control hazards in addition to a cache miss rate of 0.11% and 0.02% for the data and instruction caches respectively.

II. Background

RISC-V was developed in 2010 at the University of California, Berkeley, and is based on the 'Reduced Instruction Set Computer' (RISC) microprocessor architecture.

The execution of a RISC-V instruction involves 5 separate stages: the instruction fetch, instruction decode, execute, memory operation and register write back. In a non-pipelined processor, these processes run sequentially and are included within the critical path, this means that assuming a single clock period is sufficiently long enough for all of these processes to propagate, the total time taken for a single cycle is equal to the sum the time taken within each process. Instruction pipelining aims to address this issue.

In computer architecture, pipelining is the implementation of instruction-level parallelism within a single processor [2]. This process aims to utilise of each part of the processor in parallel, reducing their idle time while also increasing the maximum frequency at which the processor can be ran at and is achieved by dividing incoming instructions into sequential steps in which each pipeline stage of the processor operates on that individual instruction and prepares it for the next stage. As a result, the data propagation path within a given cycle is shortened allowing for the overall maximum clock frequency to significantly increase as the maximum time for a clock cycle within this process depends only on the longest path within a given pipeline stage.

In an ideal situation in that there are no data or control hazards and an infinite number or instructions, the components within the processor can be used in parallel and the total cycles per instruction (CPI) will be equal to 1. Hence, the overall execution time should be lower than the original non-pipelined design as the CPI is equivalent but the maximum frequency of the processor is higher.

In modern computing, data is generally stored in four locations, registers, caches, memory and storage with each of the time to read/write and cost of each ranging significantly. Specifically, the time taken for data to be read/written within each of these locations increase exponentially with their costs also decreasing exponentially with cache being between 10 to 100 times faster than DRAM. Hence, to obtain a higher performance speed, storing and accessing data within the cache so that future requests for that data can be obtained faster.

III. Architecture

A. Program Characterisation

The program which the processors were tested against was a program which performed a loop where every 50th number between 0 and 999 (inclusively) had a fast square root algorithm performed on it . For the testing of the processors, only the sub-processes and its related functions were used within the testing of the processors.

Viewing either the definitions of the functions within 'program.c' (appendix fig. 15) and the assembly language translation of the program (appendix fig. 16), it is evident that the program performs a significant number of jumps and branches with some data hazards due to the significant utilisation of loops and if statements within the loops. Given this, both data and control hazards must be detected and prevented by the 2-stage pipeline processor to ensure the expected output will be obtained.

Another characteristic within the program is the 'for loop' within the 'sqrttable' sub-process. This sub-process uses the

memory to read and write the parameters of the 'for loop' which are also the input arguments of the 'isqrt' function.

Architecture of the processor

1) Original NERV processor Design: As stated previously, the execution of a RISC-V instruction involves 5 separate stages - the instruction fetch, instruction decode, execute, memory operation and register write back - where the both the memory operation and register write back stages only occur during certain instruction types such as STORE or Immediate instructions. Originally, these stages operated sequentially with an instruction being completed in each clock cycle. As a consequence, the critical path of the processor included all 5 stages.

It should also be noted that in the original design, the load instruction takes two cycles to complete as its enable signal is checked on a positive clock edge and data is then written in the next cycle.

Two-stage pipelined processor: As the goal of pipelining is to increase the maximum frequency of the processor by reducing the length of the critical path within across the processor (i.e. reduce the propagation delay), the position of each pipeline register within a 2 stage pipelined processor will significantly influence the propagation delay within either stages and therefore influence the clock period. Within an ideal scenario, each stage within a pipeline process should have equivalent propagation delay as the overall clock period of the processor depends on the maximum delay within each stage.

Now determining the placement of these pipeline registers, as there are a total of 5 stages within the instruction execution process of a RISC-V processor, there were 4 potential sections to place said pipeline registers. As placing the pipeline processors between the instruction fetch and instruction decode stages in addition to the potential placements after the ALU, the only potential placement of the pipeline registers for the 2 stage pipelined processor was between the instruction decode and execute stages.

The addition of the pipeline registers between these stages both data and control hazards can potentially occur. These come in the form of Read After Write (RAW) and delays slots after completing jump/branch instructions.

Incorporating Hardware Interlocks: Incorporating hardware interlocks (appendix fig. 14) into the previous design allows the processor to automatically resolve both data and control hazards it can detect by introducing an additional control signal at each pipeline register which gave the new signal flow diagram shown above. The only type of data hazards which can occur within this type of processor is a RAW. This occurs when an instruction reads a value from a register before the processor can correctly update that value, potentially leading to calculation errors within the execution stage. To prevent this from occurring, incorporating hardware interlocks into the design allows the processor to automatically stall the program by keeping the program counter the same and 'killing' the instruction within the first stage to allow for the register to be updated and the correct value can be obtained within the next cycle. Specifically, the stall condition to detect the hazard before it occurs is:

```
stall = ((insn_rs1 == insn_rd_EX) && \
    write_enable && read_en_rs1) ||  \
    ((insn_rs2 == insn_rd_EX) &&     \
    write_enable && read_en_rs2);
```

This allows the instruction within its second pipeline stage (EX stages and so on) to write while stalling the instruction within the first pipeline stage so that within the next cycle, the program counter and the instruction within the first pipeline stage remains the same while a bubble (i.e. NOP) is created and executed within the second pipeline stage. An example of this occurring is demonstrated with appendix figure 3.

In addition to the processor detecting and resolving data hazards, compatibility to detect and resolve control hazards by stalling/killing the process. As both conditional branches and unconditional jumps are computed within the second pipeline stage, an additional 'delay' instruction is stored and prepared within the first pipeline stage within this design. When not addressed, this causes control hazards as the instruction after a jump or branch is still executed. To resolve this issue, a control signal from the branch and jump logic is used to 'kill' the instruction within the first pipeline stage of the design by replacing the instruction with a NOP.

A flaw within this design is the fact that it introduces an additional instruction into the program, essentially increasing the CPI for each hazard detected. As the execution time of a program is proportional to both the CPI and cycle period, the execution time of the program can potentially increase compared to the original design depending on the number of hazards experienced.

Incorporating Forwarding: Incorporating forwarding into the design, results obtained within the execution stage can immediately be used within the next cycle providing an alternative solution to data hazards. Using this method most operations no longer need to be stalled when a potential data hazard is detected and therefore, the number of cycles per instructions can reduce and become closer to 1.

B. Cache

Cache operates by exploiting two predictable properties of memory references, temporal locality and spacial locality. Temporal and spatial locality occurs since for each location referenced, that location and locations near it is likely to be referenced again in the near future. This allows the reading of data and instructions into memory to be significantly faster than processors which do not utilise cache. For the program tested within this report, as this program utilises a large number of 'while' and 'for' loops in addition to if statements within these loops to obtain each output, the main memory reference property in which caches exploit is temporal locality with spacial locality not being used as frequently, if at all.

Caches have three parameters which influence its performance - cache size (block size), associativity (ways) and line size (set) - with each parameter having a different impact on

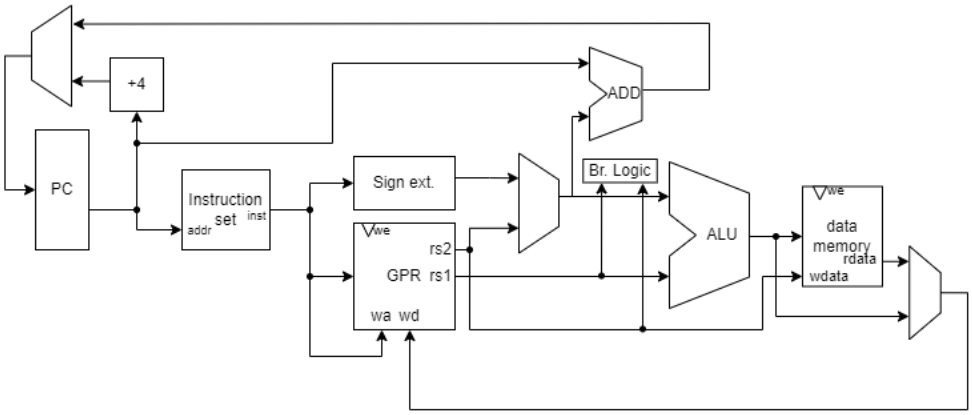

Fig. 1. Original NERV architecture

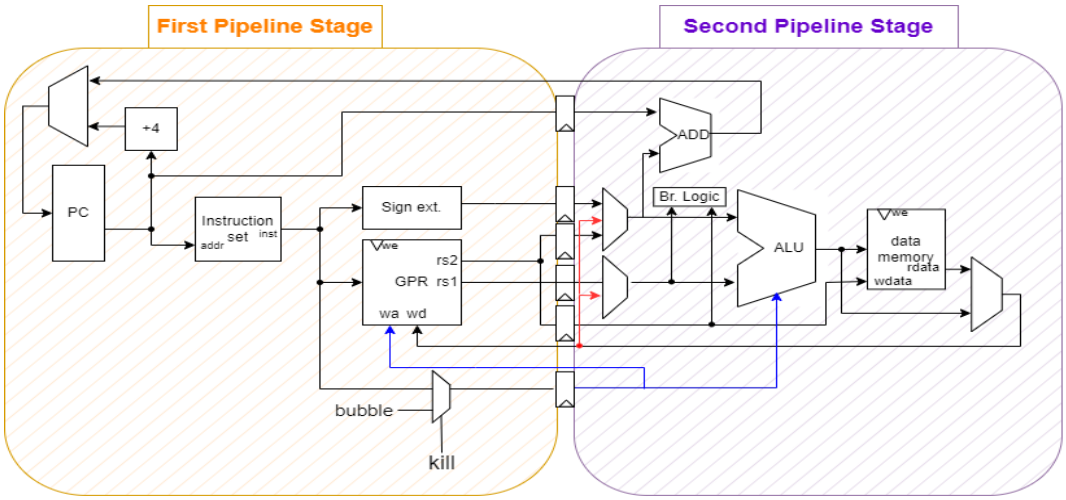

Fig. 2. 2 stage pipeline architecture with hardware interlock and forwarding features

the performance. Increasing the cache size reduces the capacity and conflict misses but increases the hit time whereas higher associativity leads to reduced conflict misses and longer hit times and larger line size causes reduced compulsory and capacity misses with an increase in conflict misses and miss penalty. Due to these factors and the innumerable possible combinations of these parameters, to obtain the best cache organisation optimised for this processor and program a simulation was used. This simulation calculates and minimises the product between the Average Memory Access Time (AMAT) and cost to find the most optimal combination.

IV. Results

Two separate tests were simulated for both the NERV processor and 2-stage pipelined processor with 'Test all' testing all the sub-processes (i.e. sqrttable and isqrt functions) and 'Test isqrt' which only tested the isqrt function with an input of 400. Simulations of the original NERV processor and 2-stage pipelined processor completing a modified version of program-2 were made and 5 main data points were collected: the cycle period of the processor, the outputs of the sub-processes, the number of cycles it took to complete the program, the execution time and the normalised execution time relative to the original NERV processor.

Original Design

Evidence of information shown below in appendix fig. 11 Outputs shown in appendix fig. 4 with example wave diagram out output in appendix fig. 6

Cycle Period: 27.46 ns

TABLE I
ORIGINAL TEST RESULTS

Test name	Number of Cycles	Execution time (s)	Normalised Execution time
Test all	1318	3.61889e-05	1
Test isqrt	66	1.81219e-06	1

Two-stage pipelined processor

Evidence of information shown in below appendix fig. 12 Outputs shown in appendix fig. 4

Cycle Period: 21.24ns
Geometric mean execution time: 0.99

TABLE II
PART 3 TEST RESULTS

Test name	Number of Cycles	Execution time (s)	Normalised Execution time (s)
Test all	1699	3.60875e-05	0.997
Test isqrt	84	1.78419e-06	0.985

Cache optimisation

Using the python script shown in appendix fig. 17, the output (appendix fig. 10) minimising the product of the AMAT and the memory cost, it was found that the best combination for the Data and Instructions cache were both two-way set associative with a set size of 1 and a block size of 512. Given this, the miss rate for either caches were found to be 0.11% and 0.02% respectively

V. DISCUSSION

A. Results discussion

Observing the results above, it can be noted that with each implementation, cycle period, number of cycles for each program and hence the normalised execution time of each also changed. This subsection discusses how the improvements made between each stage lead to the results being produced.

1) Hardware Interlocks: As stated within the architecture section, hardware interlocks are used within a multi-staged pipeline processor to detect and prevent hazards from occurring. Within this case, the main hazards which occur are Read-After-Write data hazards and control hazards. When comparing the difference in the number of cycles computed for the NERV processor and the two-stage pipeline processor within the results, it is evident that the number of cycles the two-stage pipeline processor had to complete was significantly higher. This was due to the detection of control hazards leading to the killing of the instruction within the first stage and an automatic insertion of a 'bubble' into the instruction pipeline which takes its place.

Evidence of this occurring is demonstrated within appendix fig. 8 in which the stall signal is raised within the first stage of the pipeline processor which kills and implements a bubble within the second instruction pipeline stage ('insn_ex') within the next clock cycle.

2) Forwarding: With the addition of forwarding, the processor is able to handle data hazards without the need of hardware interlocks. Seen within appendix fig. 9, after detecting the data hazard, instead of stalling the instruction pipeline and substituting a bubble within the first stage, a forwarding flag is raised for the rs1 or rs2 value which caused the data-hazard, allowing the computed value to be used within the next cycle whilst the register itself is updating. As a result of this implementation, all data hazards experienced by the processor can be automatically computed and handled without the use of hardware interlocks, decreasing the number of stalls in which data hazards usually present to a multi-stage pipelined processor, bringing the CPI of the program closer to 1.

B. Caching

As described in the architecture, due to the numerous combinations of cache parameters, finding the most optimal cache size was not designed but rather simulated and tested against the program. In doing so, the most optimal cache configuration for this program was found to be a two-way set associative with a set size of 1 and a block size of 512 for both the data and instruction cache.

C. Areas of Improvement

As stated previously, the main flaw when implementing instruction pipelining within a processor is the appearance of data, control and structural hazards. With the implementation of hazard detection capabilities which stall or kill instructions, the hazards are automatically prevented by the processor but the CPI is reduced as the operation of the program is delayed for each hazard detected. Given that forwarding handles the data hazards appropriately, the next major improvement would be the addition of branch prediction. In doing so, the number of control hazards can be reduced, bringing the CPI closer to 1.

The next improvement made would be the calculation of the Average Memory Access Time (AMAT) of the processor. As the formula used within the calculation was that it takes 1 cycle period to obtain the data from memory and 10 cycles when it misses, the formula is very general and does not fully consider the effects of increasing the block size and the associativity of the cache. This means that the calculation for the optimal combination is not fully accurate.

VI. CONCLUSION

This report illustrates the effects of the implementation of a 2 stage pipeline RV32I processor whilst comparing its results to a standard Naive Educational RISC-V processor. Simulating each processor using SystemVerilog, it was found that the proposed prototype 2 stage pipeline processor (with hardware interlock and forwarding capabilities) had a performance increase of 5.78% compared to the original processor.

This report illustrates the hazard detection within a two-stage pipeline RV32I processor and suggests an optimal cache configuration for the specified fast square root program. Using SystemVerilog and other computer simulation scripts, this report demonstrates automatic hardware interlock stalling when encountering control hazards and the forwarding of values when encountering data hazards in addition to the optimal cache configurations

REFERENCES

[1] Wikipedia contributors. (2022, September 25). RISC-V. In Wikipedia, The Free Encyclopedia. Retrieved 12:09, October 16, 2022, from https://en.wikipedia.org/w/index.php?title=RISC-V&oldid=1112336599

[2] Wikipedia contributors. (2022, August 17). Instruction pipelining. In Wikipedia, The Free Encyclopedia. Retrieved 12:10, October 16, 2022, from https://en.wikipedia.org/w/index.php?title=Instruction_pipelining

APPENDIX

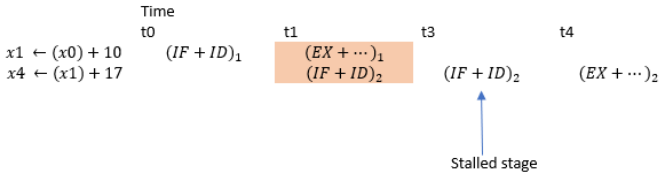

Fig. 3. 2 stage pipeline stall example

```
info: answer found:          7
info: answer found:         10
info: answer found:         12
info: answer found:         14
info: answer found:         15
info: answer found:         17
info: answer found:         18
info: answer found:         20
info: answer found:         21
info: answer found:         22
info: answer found:         23
info: answer found:         24
info: answer found:         25
info: answer found:         26
info: answer found:         27
info: answer found:         28
info: answer found:         29
info: answer found:         30
info: answer found:         30
x10=30, cycles=1318
```

Fig. 4. Console output of test_all.s using NERV processor

```
info: answer found:          7
info: answer found:         10
info: answer found:         12
info: answer found:         14
info: answer found:         15
info: answer found:         17
info: answer found:         18
info: answer found:         20
info: answer found:         21
info: answer found:         22
info: answer found:         23
info: answer found:         24
info: answer found:         25
info: answer found:         26
info: answer found:         27
info: answer found:         28
info: answer found:         29
info: answer found:         30
info: answer found:         30
x10=30, cycles=1699
```

Fig. 5. Console output of test_all.s using two-stage pipelined processor

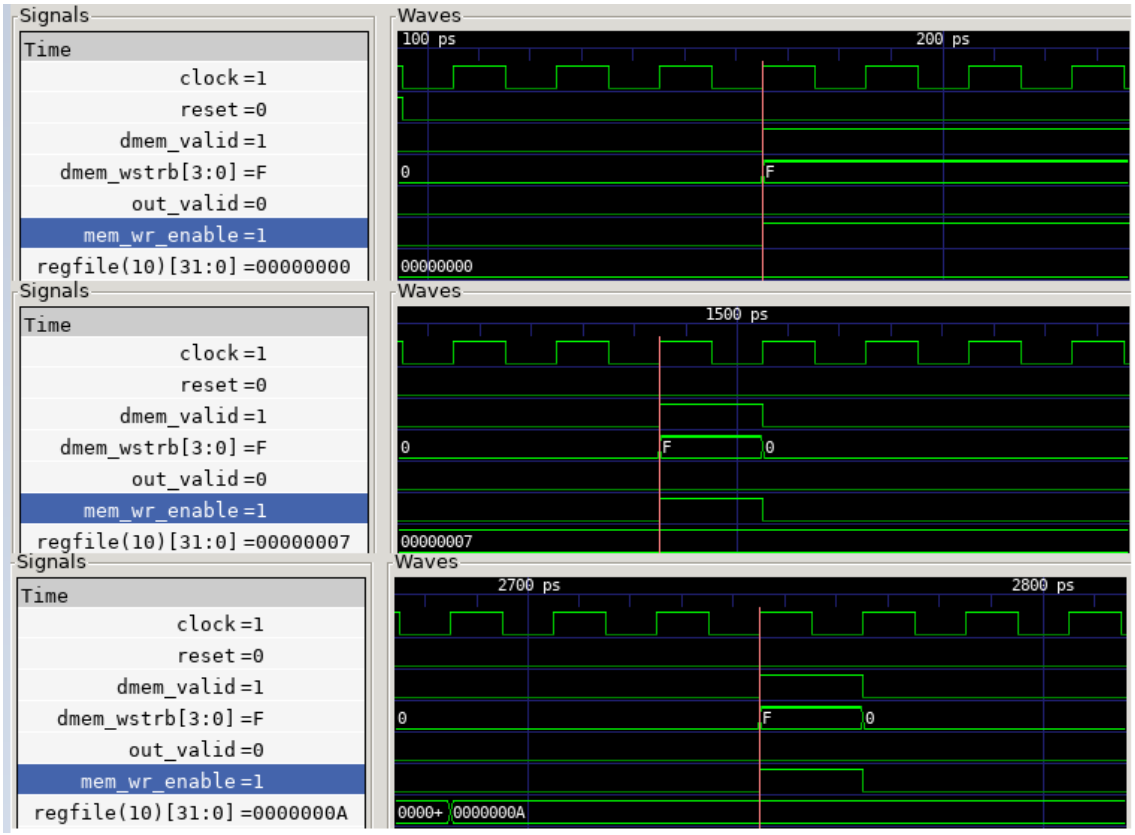

Fig. 6. First three return answers using NERV processor

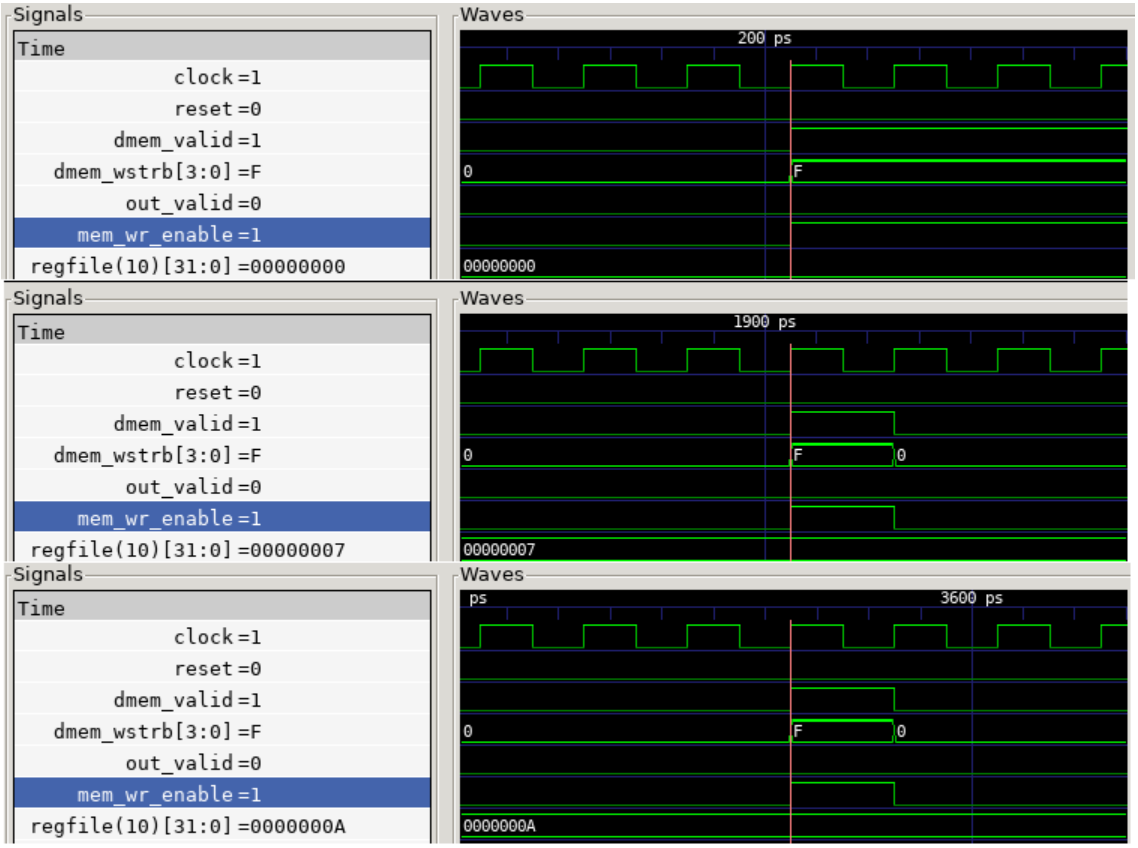

Fig. 7. First three return answers using two-stage pipelined processor

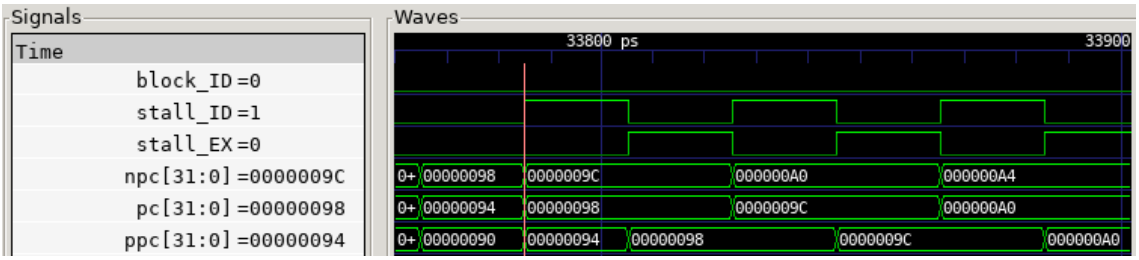

Fig. 8. Automatic hardware interlock functionality within two-stage pipelined processor

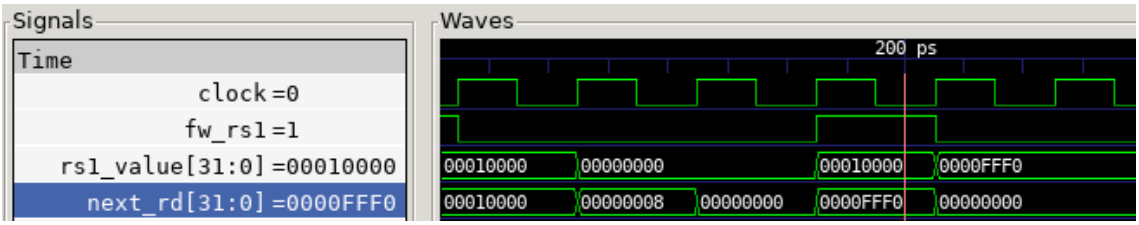

Fig. 9. Automatic forwarding functionality within two-stage pipelined processor

```
Best combinations when minimising miss rate:
$D:     Best miss rate: 0.11%, Best Combination: (1, 1, 512)
$I:     Best miss rate: 0.02%, Best Combination: (1, 1, 512)
---------------------------------------
Best combinations when minimising score:
$D:     Best score: 0.1011, Best Combination: (1, 2, 512) with miss rate: 0.11%
$I:     Best score: 0.10020000000000001, Best Combination: (1, 2, 512) with miss rate: 0.02%
```

Fig. 10. python script finding optimal cache parameters results

```
elec3608@6891137235b4:~/elec3608-lab/FinalExam/Section_2$ make result
python extime.py nerv-pnr.log *.result
Test summary
period: 2.7457440966501922e-08
test_all.result: x10=30 cycles=1318 extime=3.6188907193849535e-05 normextime=1.0
test_isqrt.result: x10=20 cycles=66 extime=1.8121911037891268e-06 normextime=1.0
Geometric mean=1.0
```

Fig. 11. Execution time of original NERV processor

```
elec3608@6891137235b4:~/elec3608-lab/FinalExam/Section_2$ make result
python extime.py nerv-pnr.log *.result
Test summary
period: 2.1240441801189465e-08
test_all.result: x10=30 cycles=1699 extime=3.60875106202209e-05 normextime=0.9971981310989719
test_isqrt.result: x10=20 cycles=84 extime=1.784197111299915e-06 normextime=0.9845524059627714
Geometric mean=0.9908550949533801
```

Fig. 12. Execution time of Two-Stage Pipeline Processor with hazard detection and forwarding

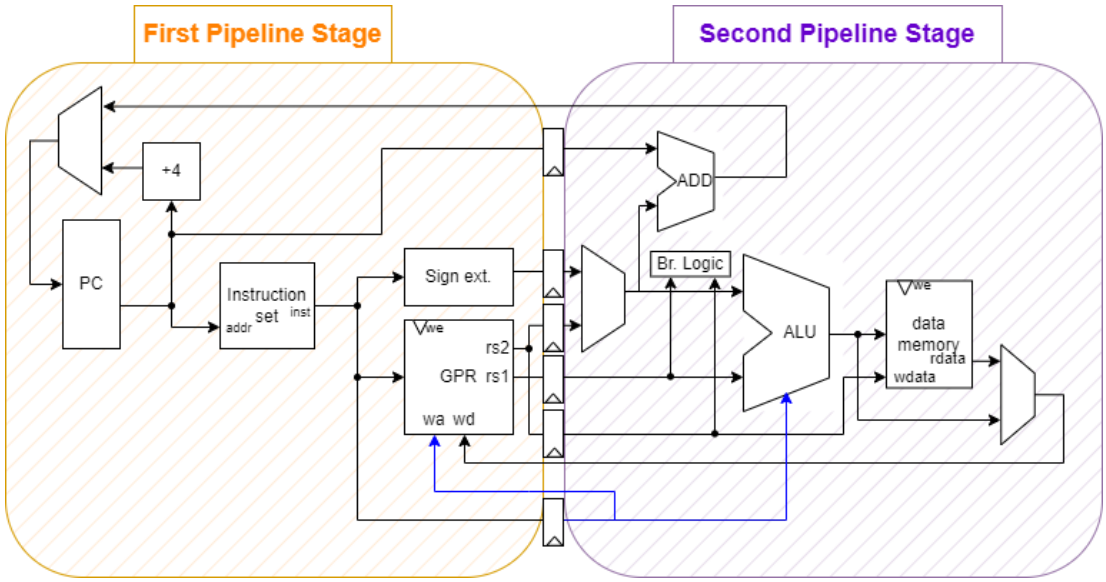

Fig. 13. 2 stage pipeline architecture

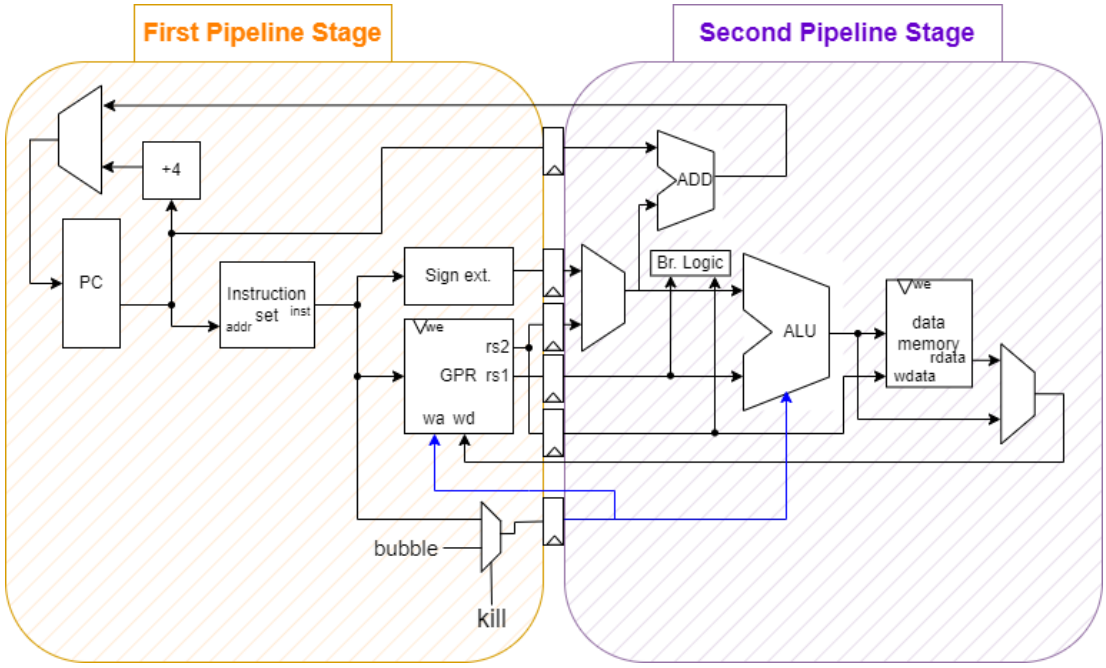

Fig. 14. 2 stage pipeline architecture with hardware interlock feature

```c
#include <stdlib.h>
#include <stdio.h>

/**
 * \brief    Fast Square root algorithm
 *
 * Fractional parts of the answer are discarded. That is:
 *      - SquareRoot(3) --> 1
 *      - SquareRoot(4) --> 2
 *      - SquareRoot(5) --> 2
 *      - SquareRoot(8) --> 2
 *      - SquareRoot(9) --> 3
 *
 * \param[in] a_nInput - unsigned integer for which to find the square root
 *
 * \return Integer square root of the input value.
 */

#define MAXN    999
#define STEP    50

unsigned    table[MAXN];

unsigned isqrt(unsigned a_nInput)
{
    unsigned op  = a_nInput;
    unsigned res = 0;
    unsigned one = 1uL << 30;
    // The second-to-top bit is set: use 1u << 14 for uint16_t type;
    // use 1uL<<30 for unsigned type


    // "one" starts at the highest power of four <= than the argument.
    while (one > op)
        one >>= 2;

    while (one != 0)
    {
        if (op >= res + one)
        {
            op = op - (res + one);
            res = res +  2 * one;
        }
        res >>= 1;
        one >>= 2;
    }
    return res;
}
```

```c
void
sqrttable()
{
    int i;

    for (i = 50; i < MAXN; i += STEP) {
        table[i] = isqrt(i);
    }
}

int
main()
{
    unsigned    i = 1234;
    unsigned    r;

    sqrttable();
    for (i = 0; i < MAXN; i += STEP)
        printf("isqrt(%d)=%d\n", i, table[i]);

    return 0;
}
```

Fig. 15. Code listing: program.c

```
    .file   "program.c"
    .option nopic
    .attribute arch, "rv32i2p0"
    .attribute unaligned_access, 0
    .attribute stack_align, 16
    .text
    .align  2
    .globl  isqrt
    .type   isqrt, @function
    li sp, 65536        # moving stack pointer to the end of the memory
    j sqrttable         # starting at the sqrttable subprocess
isqrt:
    mv  a4,a0           # a0 is op, a4 is a_nInput
    li  a5,1073741824   # a5 is one
    bgeu    a0,a5,.L7   # jump if op > one
.L3:
    srli    a5,a5,2     # one shift right by 2
    bltu    a4,a5,.L3   # while loop check
    beq a5,zero,.L11    # jump if one == 0
.L2:
    li  a0,0            # a0 is now made into res
    j   .L6
.L7:
    li  a5,1073741824
    j   .L2
.L5:
    srli    a0,a0,1     # res shift right by 1
    srli    a5,a5,2     # one shift right by 2
    beq a5,zero,.L12    # while loop condition
.L6:
    add a3,a0,a5        # temp = res + one
    bgtu    a3,a4,.L5   # jump if temp > op (ie. op >= temp) (skips if statement)
    sub a4,a4,a3        # op -= res + one;
    add a0,a3,a5        # res = temp + one; (i.e. res = res + 2*one)
    j   .L5
.L12:
    ret
.L11:
    mv  a0,a5
    ret
    .size   isqrt, .-isqrt
    .align  2
    .globl  sqrttable
    .type   sqrttable, @function
```

```
sqrttable:
    addi    sp,sp,-16
    sw  ra,12(sp)
    sw  s0,8(sp)
    sw  s1,4(sp)
    sw  s2,0(sp)
    lui s1,%hi(table+200)
    addi    s1,s1,%lo(table+200)
    li  s0,50
    li  s2,1000
.L14:                   # for loop
    mv  a0,s0
    call    isqrt
    sw  a0,0(s1)
    addi    s0,s0,50
    addi    s1,s1,200
    bne s0,s2,.L14
    lw  ra,12(sp)
    lw  s0,8(sp)
    lw  s1,4(sp)
    lw  s2,0(sp)
    addi    sp,sp,16
    ebreak              # exit before returning to main
    jr  ra
    .size   sqrttable, .-sqrttable
    .section    .rodata.str1.4,"aMS",@progbits,1
    .align  2
```

Fig. 16. Code listing: test_all.s

```python
import os

# alter these values to change depth/range of the scan
b_scan = 10      # note: scans block sizes 2**3, 2**4, ... 2**(b_scan-1)
s_scan = 10      # similar to above but from 2**0, ...
w_scan = 10      # scans all values from 1:w_scan-1

command0 = "spike --isa=rv32i --ic=x:y:z --dc=x:y:z  \
 /opt/riscv/riscv32-unknown-elf/bin/pk program"

# values based on miss rate
bst_ms_r_D = 100
bst_ms_r_I = 100
bst_comb_D = (0, 0, 0)
bst_comb_I = (0, 0, 0)

# values based on cost and missrate relationship (AMAT)
# suppose memory cost = ($0.1 + $0.0001 * b_size * s_size * w_size)
# and the hit time is 1 cycle and the miss penalty is 10 cycles,
# we want to minimise AMAT * memory cost (i.e. score)
bst_score_D = (1.0, 1.0)
bst_score_I = (1.0, 1.0)
bst_comb_D_s = (0, 0, 0)
bst_comb_I_s = (0, 0, 0)

calc_score = lambda ms_r_pc, s, w, b: (1 + (float(ms_r_pc)/100 * 10)) * (0.1 + 1e-4*b*s*w)
for b_size in range(3, b_scan):
    command1 = command0.replace("z", str(2**b_size))
    for s_size in range(s_scan):
        command2 = command1.replace("x", str(2**s_scan))
        for w_size in range(1, w_scan):
            command3 = command2.replace("y", str(2**w_size))
            res = os.popen(command3).read().split('\n')
            res = [x for x in res if '%' in x]
            if len(res) != 2:
                raise Exception(f"unexpected number of outputs in scan, we got:\n{res}")
            D_pc_val = float(res[0][-7:-2].strip())
            I_pc_val = float(res[1][-7:-2].strip())

            # comparing to current best miss rate (and potentially updating)
            if D_pc_val < bst_ms_r_D:
                bst_ms_r_D = D_pc_val
                bst_comb_D = (2**s_size, w_size, 2**b_size)
            if I_pc_val < bst_ms_r_I:
                bst_ms_r_I = I_pc_val
                bst_comb_I = (2**s_size, w_size, 2**b_size)

            # comparing to best scores (and potentially updating)
            D_score = calc_score(D_pc_val, s_size, w_size, b_size)
            I_score = calc_score(I_pc_val, s_size, w_size, b_size)
            if (D_score < bst_score_D[0]):
                bst_score_D = (D_score, D_pc_val)
                bst_comb_D_s = (2**s_size, 2**w_size, 2**b_size)
            if (I_score < bst_score_I[0]):
                bst_score_I = (I_score, I_pc_val)
                bst_comb_I_s = (2**s_size, 2**w_size, 2**b_size)

    print(f"Scanned all for block size {2**b_size}")
print("----------------------------------------")
print("Best combinations when minimising miss rate:")
print(f"$D:\tBest miss rate: {bst_ms_r_D}%, Best Combination: {bst_comb_D}")
print(f"$I:\tBest miss rate: {bst_ms_r_I}%, Best Combination: {bst_comb_I}")
print("----------------------------------------")
print("Best combinations when minimising score:")
print(f"$D:\tBest score: {bst_score_D[0]}, Best Combination: {bst_comb_D_s} with miss rate: {bst_score_D[1]}%")
print(f"$I:\tBest score: {bst_score_I[0]}, Best Combination: {bst_comb_I_s} with miss rate: {bst_score_I[1]}%")
```

Fig. 17. Code listing: Optimal Cache combination python script

```
// read-enable system capture for stall condition
if (insn_rd_EX) // making sure it is not a nop or save to zero
case (insn_opcode)
    OPCODE_JALR:    begin read_en_rs1 = 1; end                  // rs1
    //OPCODE_BRANCH:    begin read_en_rs1 = 1; read_en_rs2 = 1; end // rs1 and rs2
    OPCODE_LOAD:    begin read_en_rs1 = 1; end                  // rs1
    // OPCODE_STORE:    begin read_en_rs1 = 1; read_en_rs2 = 1; end // rs1, rs2
    // OPCODE_OP_IMM:   begin read_en_rs1 = 1; end                  // rs1
    // OPCODE_OP:       begin read_en_rs1 = 1; read_en_rs2 = 1; end // rs1, rs2
    default:;
endcase

// implementing forwarding
if (!block_EX)
case (insn_rd_EX)
    insn_rs1: fw_rs1 = 1;
    insn_rs2: fw_rs2 = 1;
    default:  begin fw_rs1 = 0; fw_rs2 = 0; end
endcase
```

```
// defining stall condition
if (!stall_ID) stall_ID = ((insn_rs1 == insn_rd_EX) && (next_wr || mem_rd_enable_q) && read_en_rs1) || ((insn_rs2 == insn_rd_EX) && (next_wr || mem_rd_enable_q) && read_en_rs2);

if (j_b_check) block_ID = 1; // stops jump from happening for items in delay slot
```

```
// PIPELINE REGISTERS
always @(posedge clock) begin
    // pipelining pc
    ppc <= ($signed(pc) > 0) ? pc : 0;

    // pipelining sign-ext
    imm_i_sext <= imm_i_sext_ID;
    imm_s_sext <= imm_s_sext_ID;
    imm_b_sext <= imm_b_sext_ID;
    imm_j_sext <= imm_j_sext_ID;

    // add pipeline for control signals
    // i.e. for insn and so on, note: only need to pipeline for control signal
    //      not everything else as they will be individually pipelined so that rd1 and so on is chosen still
    insn_EX <= (!stall_ID) ? insn : 32'b00010011;; // nop bubble when stalled

    // add pipeline for rd1 and rd2 (and forwarding)
    rs1_value <= (fw_rs1 & !block_EX) ? next_rd : rs1_value_ID;
    rs2_value <= (fw_rs2 & !block_EX) ? next_rd : rs2_value_ID;

    block_EX <= block_ID;
    stall_EX <= stall_ID;
end
```

Fig. 18. Code listing: pipeline registers, stall condition and forwarding implementation (1)

```
assign imem_addr = (trap || stall_ID) ? imem_addr_q : npc;
    always @(posedge clock) begin
        reset_q <= reset;
        trapped_q <= trapped;

        // increment pc if possible
        if (!trapped && !reset && !reset_q) begin
            if (illinsn)
                trapped <= 1;
            if (!stall_ID) begin
                pc <= npc;
            end
            // update registers from memory or rd (destination)
            if (mem_rd_enable_q || next_wr) begin
                if (next_wr) $write(""); //$display("x%d <- %h \t (ppc = %d)", insn_rd_EX, next_rd, ppc);
                if (mem_rd_enable_q) $write(""); //$display("x%d <- %h", mem_rd_reg_q, next_rd);
                regfile[mem_rd_enable_q ? mem_rd_reg_q : insn_rd_EX] <= mem_rd_enable_q ? mem_rdata : next_rd;
            end

            if (mem_wr_enable && insn_rs2_EX == 10) $display("info: answer found:%d", rs2_value);
            x10 <= regfile[10];
        end

        // reset
        if (reset || reset_q) begin
            pc <= RESET_ADDR - (reset ? 4 : 0);
            trapped <= 0;
        end
    end
```

Fig. 19. Code listing: pipeline registers, stall condition and forwarding implementation (2)