Onn Rengingad 304845951
Bar Ben Ari 204034284
Barak Levy 311431894

Lab 5 - Advanced Computer Architecture

1. **Harvard architecture (Question 1):**

a. Advantages:
   1) In this architecture execution time is faster.
   2) Separates between RAM for data and ROM for instruction – you can access both in the same cycle.
   3) Decrease structural hazards.

b. Disadvantages:
   1) Complex to analyze and implement.
   2) Expensive – requires more physical space.

2. **Structural and Data Hazards (Question 2):**

a. Structural hazards:
   1) Description: These hazards occur when two instructions attempt to access the same hardware resource (either for reading or writing) at the same time, and hence the CPU needs to stall one of the instructions. Thus, the two instructions will access the resource in consecutive cycles - serially, instead of parallelly.
   2) Solution: Compare the store and load registers of the instructions and stall one of the instructions according to the correct run of the program so that we won't perform both instructions in the same clock cycle. The main example for such hazard, in the Harvard pipelined SP implementation for the current ISA, will be the execution of an 'ST' followed by an 'LD' operation. Since 'ST' accesses the SRAMd for writing operation in stage 'EXEC1', and 'LD' accesses the SRAMd for reading operation in stage 'EXEC0', when 'LD' is executed immediately after 'ST', then 'LD' will reach 'EXEC0' and 'ST' will reach 'EXEC1' at the **same cycle**, causing the structural hazard, when the hardware resource is the SRAMd (SRAM Data as implemented in the Harvard implementation).

b. Data hazards:
   1) Description: When either a source or a destination operands of an instruction are not available at the time expected in the pipeline e.g. attempting to read a register's data when it should have already been updated in previous instructions, and wasn't updated yet. Leading to the usage of wrong data by the

program. In the current architecture the only hazard of this kind is RAW (read after write) hazard.

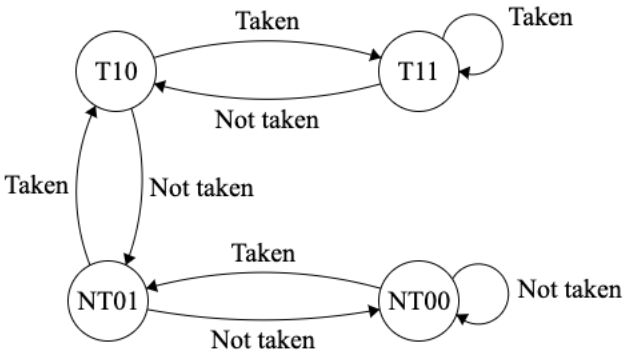
2) Solution: We validate whether the source registers are the same as any destination registers of the previous instructions that haven't finished yet, and that their destination registers weren't updated yet. If any registers of such exist as sources in the current executed instruction, then either the instruction is stalled or forwarding is used.

c. Control hazards:

1) Description: Happens due to the usage of branch instructions. While a branch operation may cause the instruction pointer to jump to a non-consecutive instruction address if the correct condition is met. If the condition is met and a jump should be taken, then the pipeline containing parts of the consecutive instructions to the branch, will proceed processing them and eventually executing them, when they shouldn't be - This is the control hazard.
2) Solution: Use branch prediction - stall and flush if we didn't predict correctly.

3. **Branches and branch prediction (Questions 3):**

We will use a 2-bit predictor. A predictor will update every branch opcode and will change according to the state machine below. The BHT (branch history table) has 20 predictors of 2-bit and each predictor is used according to the pc – for a certain pc value we use the predictor in line pc mod 20 in the table. When we encounter a hazard, we flush the pipeline.



- In each state T=taken, NT= not taken and numbers is the binary value of the predictor.

4. **Low level simulator pipeline implementation (Question 4):**

The SP files are attached within the compressed 'prog' folder.

5. **Low level testing: example.bin, mult.bin, fibo.bin (Question 5):**

The tests files are attached within the compressed 'other' folder.

6. **Speedup comparison, next generation improvements (Question 6):**

a. Speed up of each test:

1) Example test:

a) Lab 2 and 4 cycles: 366
b) Lab 5 cycles: 141
c) Speedup: 259.574%

2) Mult test:

a) Lab 2 and 4 cycles: 162
b) Lab 5 cycles: 72
c) Speedup: 225%

3) Fibo test:

a) Lab 2 and 4 cycles: 1644
b) Lab 5 cycles: 664
c) Speedup: 255.28%

b. The new simulator did not get CPI 1 as management hoped due to structural, data and control hazards that occur in the programs.

c. Ways to further improve the next generation:

1) Add more cores for paralleling programs.
2) Add a better compiler that can do loop unrolling, software pipeline and scheduling the assembly code.

Onn Rengingad 304845951
Bar Ben Ari 204034284
Barak Levy 311431894

3) Use more complicated dynamic (bigger bht or tournament predictor) and static branch predictions.

7. **DMA testing in a pipeline environment (Question 7):**

1. The DMA testing files attached within the 'other' folder.
2. The assembly code that validates the DMA operation and hazards is as follows:

```
//Includes CPU Hazards validation tests: Data, Structural, Control(every branch operation is this hazard's validation)....

//Setting registers for COPY operation
asm_cmd(ADD, 4, 1, 0, 30);   // 0: R4 = 30 -COPY source address
asm_cmd(ADD, 5, 1, 0, 1000); // 1: R5 = 1000 -COPY destination address
asm_cmd(ADD, 6, 1, 0, 1000); // 2: R6 = 1000 -COPY operation number of cells taken

//Initiate copy regime in a different thread -  (ALSO A VALIDATATION FOR Data Hazard at register 6 and register 5)
asm_cmd(COPY, 5, 4, 6, 0);   // 3: COPY - DMA copy operation is initiated:
                             // It will copy 1000 cells from address


//Begin an Accumulative-negative-sum computation in the main thread
    //Variables initiation
asm_cmd(ADD, 2, 1, 0, 50);   // 4: R2 = 50
asm_cmd(ADD, 3, 1, 0, 200);  // 5: R3 = 200
asm_cmd(ADD, 4, 0, 0, 10000);// 6: R4 = 0 (10000 in immediate isn't necessary)

//Test Structural Hazard in Harvard SP implementation - Store value followed by a load value into\from SRAMd (lines 8 & 9)
asm_cmd(ADD, 6, 0, 1, 151);  //7: R6 = 151
asm_cmd(ST, 0, 6, 1, 49);    //8: Mem[49] = R6  (The expected image in SRAMd is that starting at address 49 there will be (in HEX) 97 96 95 94...)

    //Load values and compute
asm_cmd(LD, 5, 0, 2, 0);     // 9: R5 = Mem[R2]
asm_cmd(SUB, 4, 4, 5, 0);    // 10: R4 -= R5
asm_cmd(ADD, 2, 2, 1, 1);    // 11: R2++
asm_cmd(JLT, 0, 2, 3, 9);    // 12: if R2<R3 jump to line 9


//Polling the DMA copy operation after the main Accumulative-negative-sum computation operation had finished
asm_cmd(POLL, 7, 0, 0, 0);   // 13 POLL - R[7] recieves the number of remaining cells to be copied by the DMA
asm_cmd(JNE, 0, 7, 0, 13);   // 14: if R[7]!=0 return to line 13 until COPY operation is done

//Following the termination of the DMA and Sum calculation operations, we compare the copied block of addresses
    //Setting registers for comparison operation
asm_cmd(ADD, 4, 1, 0, 30);   // 15: R4 = 30 -COPY source address
asm_cmd(ADD, 5, 1, 0, 1000); // 16: R5 = 1000 -COPY destination address
asm_cmd(ADD, 6, 1, 0, 1000); // 17: R6 = 1000 -COPY operation total number of cells

asm_cmd(LD, 2, 0, 4, 0);     // 18: R2 = Mem[R4]
asm_cmd(LD, 3, 0, 5, 0);     // 19: R3 = Mem[R5]
asm_cmd(JNE, 0, 2, 3, 27);   // 20: if R2!=R3 (meanining Mem[R4]!=Mem[R5]) than DMA copy operation failed! jumping to pc=27
asm_cmd(ADD, 4, 4, 1, 1);    // 21: R4++ (Advancing source address)
asm_cmd(ADD, 5, 5, 1, 1);    // 22: R5++ (Advancing destination address)
asm_cmd(SUB, 6, 6, 1, 1);    // 23: R6-- (Decresing the number of cells remanining to validate)
asm_cmd(JNE, 0, 6, 0, 18);   // 24: if R6!=0 than there are still copied cell that need to be validated. jumping to pc=18 to proceed validating

asm_cmd(ADD, 2, 0, 1, 1);    // 25: R2=1 -> DMA Copy & Fib operation were successful
asm_cmd(JIN, 0, 1, 0, 28);   // 26: Jumping to HALT (Indirect jump to the end)
asm_cmd(ADD, 2, 0, 0, 0);    // 27: R2=0 -> DMA Copy Failed
asm_cmd(HLT, 0, 0, 0, 0);    // 28: HALT....
```

The DMA assembly code was slightly altered to include a Structural hazard validation (in lines 7-9), and thus, the output trace files are different than the ones of experiments 2 & 4.

3. The above DMA assembly code utilizes the DMA hardware component to copy a block of data from an initial source address to an initial destination address, while

also maintaining some main operation of subtraction of elements. During its' run, in the pipelined CPU structure, various hazards are prevented:

- **Data Hazard**: Various data hazards may be detected e.g., in lines 0-3 - COPY at line 3 utilizes the data stored in registers r4, r5 & r6 that are updated in the ADD instructions at line 0, 1 & 2, respectively. By the time COPY reaches 'DEC1' stage and prepares the ALU operands, r4 may have done being updated, but the instructions updating register r5 & r6 are placed in stages 'EXEC1' & 'EXEC0' respectively, so both registers r5, r6's updated values are **forwarded** to the COPY ALU operands when it reaches stages 'DEC1'(r5), 'EXEC0'(r6) and overcomes the hazard, as needed.
  In the **instruction trace**(inst_trace.txt), at the first 4 instructions, one may see that in instruction 3(counting from 0) the COPY instruction was initiated with the desired source\destination addresses and block length values, while register r6 was updated in the **same** instruction's documentation, indicating r6's updated value was forwarded to COPY before being inserted to r6 (r5 is documented in same way, only one instruction before that).

- **Structural Hazard**: As the mentioned example detailed in Question 2, in the Pipelined SP Harvard implementation for the current ISA, the structural hazard that may occur is when a 'LD' instruction appears immediately after a 'ST' instruction - This is applied in lines 7-9. A value is first set to register r6, then a 'ST' instruction is initiated to store that value (d.151, h.97) in address 49 (right before the sequence of elements initially programmed to addresses 50-150). Right after that a 'LD' instruction is initiated to load data from address 50. The hazard is overcome by stalling the 'LD' instruction.
  The logs prove the successfulness of the operation e.g. in inst_trace.txt instruction 8 ('ST') and 9('LD') appear to have updated correctly Mem[40] and register r5, respectively, as needed. SRAMd_out.txt file shows the appearance of the value (in HEX) 97 before the value 96 at addresses 50, 1020 and 1990 (when the file starts counting its' lines from '1'), which indicates that **BOTH** the 'ST' instruction was not interrupted by the 'LD' instruction and that the DMA, during the COPY instruction, successfully copied the-now-updated Mem[49(*50 in the file*)] to the destination addresses of COPY, as desired. Lastly, in cycle_trace.txt, in cycle 13, the 'LD' instruction (9) is stalled (fetch1_active=dec1_active=0).

- **Control Hazard**: A control hazard (branch) appears whenever a branch operation is used in the code e.g., at lines 12(JLT), 14(JNE), 20(JNE), 24(JNE), 28(JIN).
  A false prediction and a successful flushing operation is notable at any iteration of the loop of lines 9-12, where the POLL instruction at line 13 (and the JNE following it) is inserted to the pipeline but then is **flushed** during the loop while r2<r3, and the branch returns to line 9 for another subtraction.

Finally, SRAMd_out text file contains all the correct values set by the COPY operation of the DMA, that copied 1000 addresses from source address 30 to destination address 1000. Overall, from 1000 to 2000 the updated values will be observable.