CS4110: Computer System Design Lab

Assignment # 4: Scalar Pipelined Processor Design

Submission Deadline: 11^{th} Oct, 2015

1 Objective:

To implement a scalar pipelined processor.

2 Processor Configuration

Our scalar processor has a 1024B instruction cache (I$) and a 1024B data cache (D$), both having a read port and a write port each and both are direct-mapped caches (the block size is 4B). Assume that both instruction and data caches are perfect, meaning there won't be any cache misses in these caches. Assume that the processor has a register file (RF) with sixteen, 16-bit registers, named R0, ..., R15. Note that R0 always stores the value "0". The register file has two read ports and a write port.

3 The Pipelined Processor

3.1 The Basic Pipeline

We consider a Reduced Instruction Set Computer (RISC) processor. The RISC architecture has the following instruction set.

The instruction set of the processor includes

- Three arithmetic instructions

```
ADD R1, R2, R3      ; R1 = R2 + R3
SUB R1, R2, R3      ; R1 = R2 - R3 done using two's complement arithmetic
MUL R1, R2, R3      ; R1 = R2 * R3
```

  Exactly one of the source operands of the arithmetic instruction can be a signed immediate operand of 4 bits stored in two's complement format.

```
ADD R1, R0, #5      ;makes R1 = 5
```

- Two data transfer instructions

```
LD R1, [Reg]        ; R1= content of the memory location whose address is specified by Reg.
SD [Reg], R1        ; [Reg] = R1
```

- Two control transfer instructions

```
JMP L1              ;Unconditional jump to location L1
BEQZ (Reg), L1      ;Jump to L1 if Reg content is zero
```

  L1 is given as an offset from current Program Counter (PC). This is called PC-relative addressing. L1 can be an 8-bit number represented in 2's complement format.

- Halt instruction

```
HLT
```

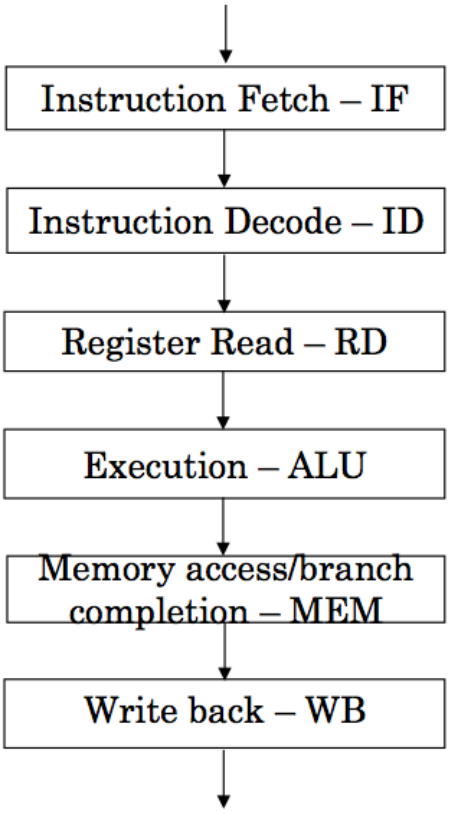

Figure 1. The six-stage instruction pipeline

There are basically six stages of instruction execution as shown in Figure **??**. Also, the instructions are assumed to be of fixed length of 2 bytes each. In a store instruction, the WB stage is non-existent. In an arithmetic instruction the MEM stage is non-existent. The processor is pipelined at the instruction level.

1. ***Instruction fetch cycle*** (IF):
   `IR ⟵ I$[PC];`
   `PC ⟵ PC + 2;`
   Operation: Send out the Program Counter (PC) and fetch the instruction from the instruction cache (IL1 cache) into the Instruction Register (IR); increment the PC by 2 to address the next sequential instruction. The IR is used to hold the instruction that will be needed on subsequent clock cycles; Register PC is updated to point to the address of the next instruction. The above describes fetching of one instruction at a time. You should fetch one instruction at any time in the scalar pipeline architecture.

2. ***Instruction decode cycle*** (ID):
   Operation: Decode the instruction; identify the opcode, source registers, and a destination register; establish the input/output dependency information. We use the following encoding: ADD – 000; SUB – 001; MUL – 010; LD – 011; SD – 100; JMP – 101; BEQZ – 110; HLT – 111. We keep one bit (ImmBit) after the opcode field to indicate whether an instruction uses immediate operand or not. If ImmBit is 1, the LSB 4 bits represent the immediate value (in 2's complement form). Since

there are 16 registers, 4 bit encoding is used for registers. R0 is encoded as 0000, R1 as 0001, etc. All the fields are at a fixed location in the instruction format.

3. ***Register read cycle*** (RD):

   ```
   A ⟵ RF[rs];
   B ⟵ RF[rt];
   Imm ⟵ sign-extended immediate fields of IR;
   ```

   Operation: Access the register file to read the registers (`rs` and `rt` are the register specifiers). The outputs of the general purpose registers are read into two temporary registers (A and B) for use in later clock cycles. The lower 4 bits of the IR are also sign extended and stored into the temporary register `Imm`, for use in the next cycle.

4. ***Execution/Effective Address cycle*** (EX):

   The ALU operates on the operands prepared in the prior cycle, performing one of the following four functions depending on the instruction type.

   - **Memory reference: (LD and ST)**

     ```
     ALUOutput ⟵ R0 + Reg;
     ```

     Operation: The ALU adds R0 with the contents of Reg fetched in earlier cycle to form the effective address and places the result into the register ALUOutput.

   - **Register-Register ALU instruction:(ADD, SUB and MUL)**

     ```
     ALUOutput ⟵ A op B
     ```

     Operation: The ALU performs the operation specified by the opcode on the values in registers A and B. The result is placed in the temporary register ALUOutput.

   - **Register-Immediate ALU Instruction:(ADD, SUB and MUL)**

     ```
     ALUOutput ⟵ A op Imm;
     ```

     Operation: The ALU performs the operation specified by the opcode on the value in register A and on the operand `Imm`. The result is placed in the temporary register ALUOutput.

   - **Branch:**

     ```
     ALUOutput ⟵ PC + (Imm << 1);
     Cond ⟵ (A == 0)
     ```

     Operation : The ALU adds the PC to the sign-extended immediate value in `Imm`, which is shifted left by 1 bit to create a 2 byte offset, to compute the address of the branch target. Register A, which has been read in the prior cycle, is checked to determine whether the branch is taken. Since we are considering only one form of branch (`BEQZ`), the comparison is against 0. Note that `BEQZ` is actually a pseudo instruction that translates to a `BEQ` with R0 as an operand. For simplicity, this is the only form of branch we consider.

5. ***Memory access cycle*** (MEM):

   ```
   LMD ⟵ D$[ALUOutput] or
   D$[ALUOutput] ⟵ B;
   ```

Operation: Access data cache, if needed. If instruction is a load, data returns from the data cache and is placed in the LMD (load memory data) register; if it is a store, then the data from the B register is written into the data cache. In either case the address used is the one computed during the prior cycle and stored in the register ALUOutput.

6. ***Write-back cycle*** (WB):

   - Register-Register ALU instruction:
     `Regs[rd] ⟵ ALUOutput;`
   - Load instruction:
     `Regs[rt] ⟵ LMD;`
     Operation: Write the result into the register file, whether it comes from the memory system (which is in LMD) or from the ALU (which is in ALUOutput); the register destination field is also in one of two positions (`rd` or `rt`) depending on the effective opcode.

3.2 Pipelining Hazards

Hazards are situations that prevent the next instruction in the instruction stream from getting executed in its designated clock cycle. Hazards may stall the pipeline. We consider data hazards and control hazards.

- **Read after write (RAW) hazards**

  Consider the instruction sequence given below.

  ```
  ADD R1, R2, R3
  SUB R4, R1, R5
  ```

  The content of R1, which is produced by the ADD instruction, is required for the SUB instruction to proceed.

- **Control hazards** arise from pipelining of branches and other instructions that change the Program Counter (PC). For e.g., in a conditional Jump instruction, till the condition is evaluated the new PC can take either the incremented PC value or the address accessed in that instruction. To avoid this we either stall the pipeline for 4 cycles or use branch predictors.

When a conflict is encountered, all the instructions before the stalled instructions need to continue and all the instructions after the stalled instruction need to be stalled.

4 Deliverables

- Assuming that each pipeline stage takes 1 cycle, execute a given program (specified in the machine language) and report 1) the CPI (clock cycles per instruction), 2) the number of stalls, and 3) the reason for each stall (ex: RAW dependency).
- Design a **graphical user interface** that shows the instruction pipeline. At each clock cycle, it should show the instructions in each stage of the pipeline.

  Interested students can implement the following things in their design for extra credit.

- Implement *operand forwarding* and *2-bit dynamic branch predictor* and compute the CPI in each of these cases.