Canvas Group Name: group\_58 Student 1: Kevin Le, [kevle@ucsc.edu](mailto:kevle@ucsc.edu) Student 2: Matthew Wark, [mtwark@ucsc.edu](mailto:mtwark@ucsc.edu)

CMPE110 HA3: Hazard Detective

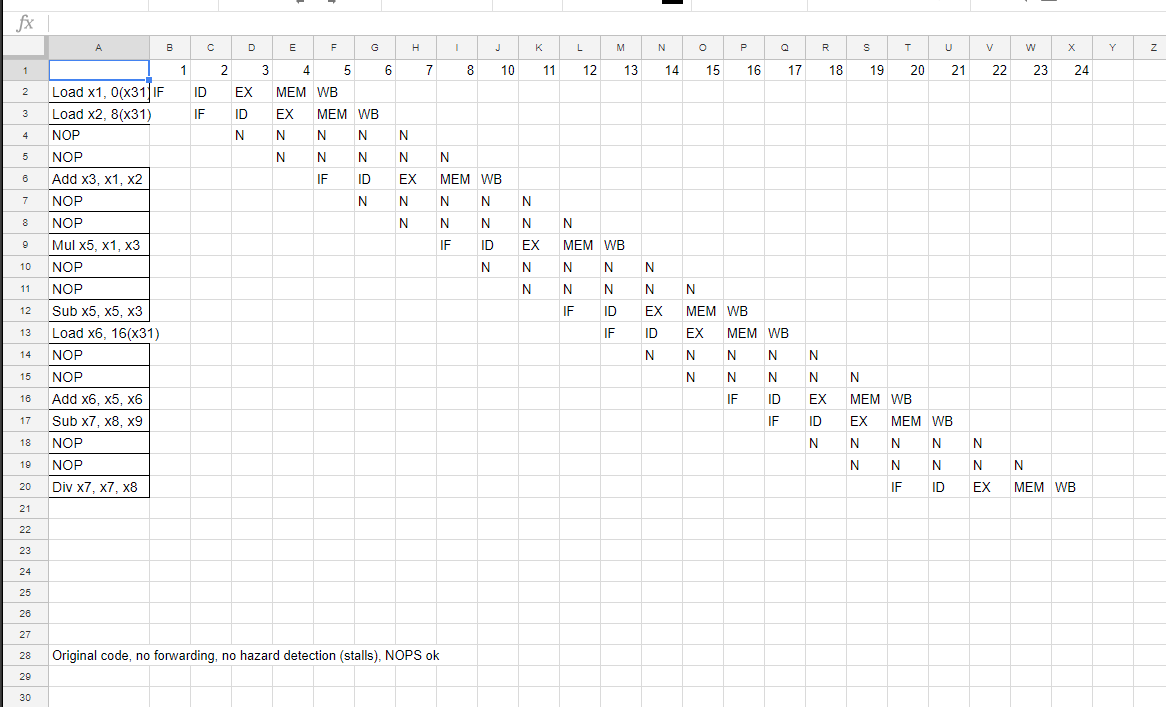
Due Date: Thursday 05/03/18

1. Consider the following instruction sequence. Determine all RAW dependencies in the code below and list them in the table **(2 Points)**

|  |  |
| --- | --- |
| Instruction | RAW Dependencies |
| Load x1, 0(x31) |  |
| Load x2, 8(x31) |  |
| Add x3, x1, x2 | RAW dependency on x1, and on x2 |
| Mul x5, x1, x3 | RAW dependency on x1, and on x3 |
| Sub x5, x5, x3 | RAW dependency on x5, and on x3 |
| Load x6, 16(x31) |  |
| Add x6, x5, x6 | RAW dependency on x6, and on x5 |
| Sub x7, x8, x9 |  |
| Div x7, x7, x8 | RAW dependency on x7 |

Figure 1: 5-stage pipeline without forwarding and hazard detection

1. Consider the 5-stage pipeline design we discussed in the lecture, shown in Figure 1, without forwarding paths and without hazard detection logic. In this case the compiler is responsible for avoiding RAW hazards. Insert the minimum number of required NOPs into the instruction stream without re-ordering instructions to avoid all hazards. Calculate the average CPI of this code, assuming a CPI of one for all instructions. Assume that registers from the register file can be written and read in the same cycle. **(4 Points)**

****

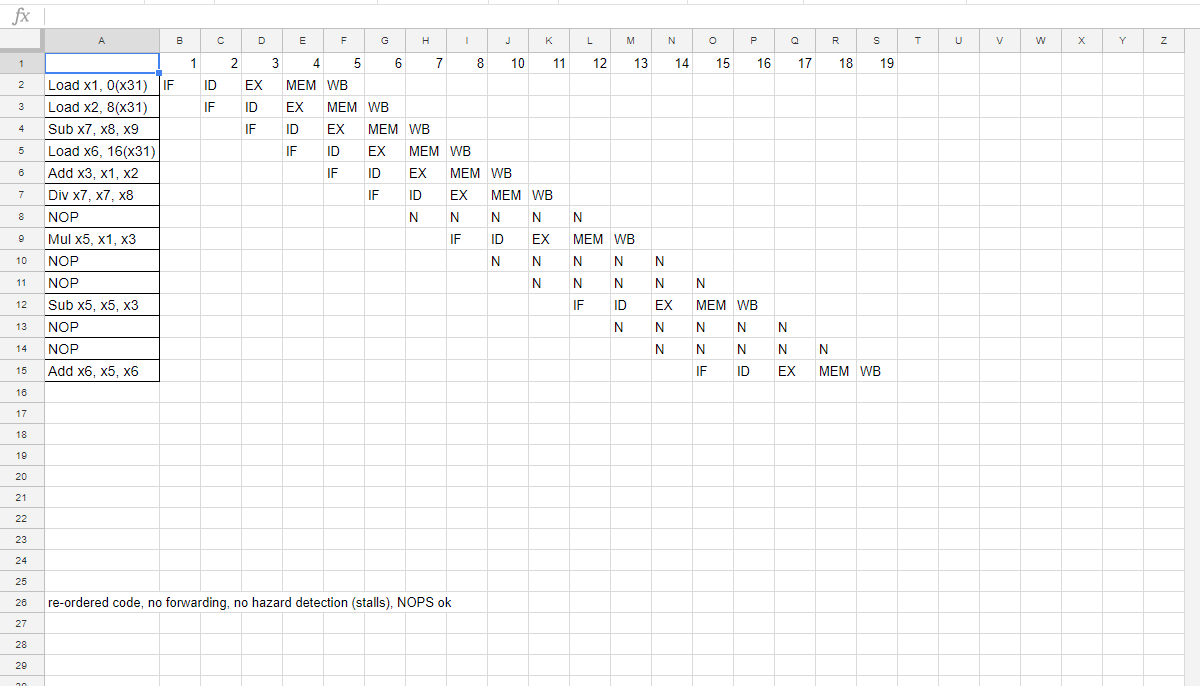
<https://docs.google.com/spreadsheets/d/1K94DBbcZT0YveJwDGUfVhVOzIB_ZbV1EXr_PjV5JDEs/edit?usp=sharing>

|  |
| --- |
| Load x1, 0(x31) |
| Load x2, 8(x31) |
| NOP |
| NOP |
| Add x3, x1, x2 |
| NOP |
| NOP |
| Mul x5, x1, x3 |
| NOP |
| NOP |
| Sub x5, x5, x3 |
| Load x6, 16(x31) |
| NOP |

24 cycles for 9 instructions

CPI of this code: 2.666666

1. Consider the 5-stage pipeline design we discussed in the lecture, shown in Figure 1, without forwarding paths and without hazard detection logic. In this case the compiler is responsible for avoiding RAW hazards. Optimize execution time by **re-ordering instructions** and then **insert the minimum number of required NOPs** into the instruction stream. Be careful to not change the output of the code when applying transformations. Calculate the average CPI of this code, **assuming a CPI of one for all instrs**. **Assume that registers from the register file can be written and read in the same cycle. (4 Points)**



<https://docs.google.com/spreadsheets/d/1K94DBbcZT0YveJwDGUfVhVOzIB_ZbV1EXr_PjV5JDEs/edit?usp=sharing>

|  |
| --- |
| Load x1, 0(x31) |
| Load x2, 8(x31) |
| Sub x7, x8, x9 |
| Load x6, 16(x31) |
| Add x3, x1, x2 |
| Div x7, x7, x8 |
| NOP |
| Mul x5, x1, x3 |
| NOP |
| NOP |
| Sub x5, x5, x3 |
| NOP |
| NOP |
| Add x6, x5, x6 |
|  |

19cycles/9ins CPI of this code: 2.111111

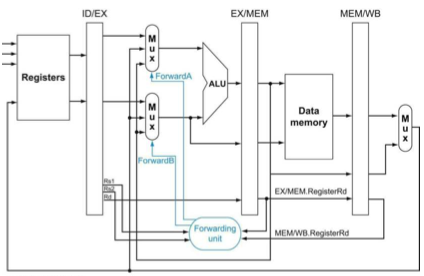
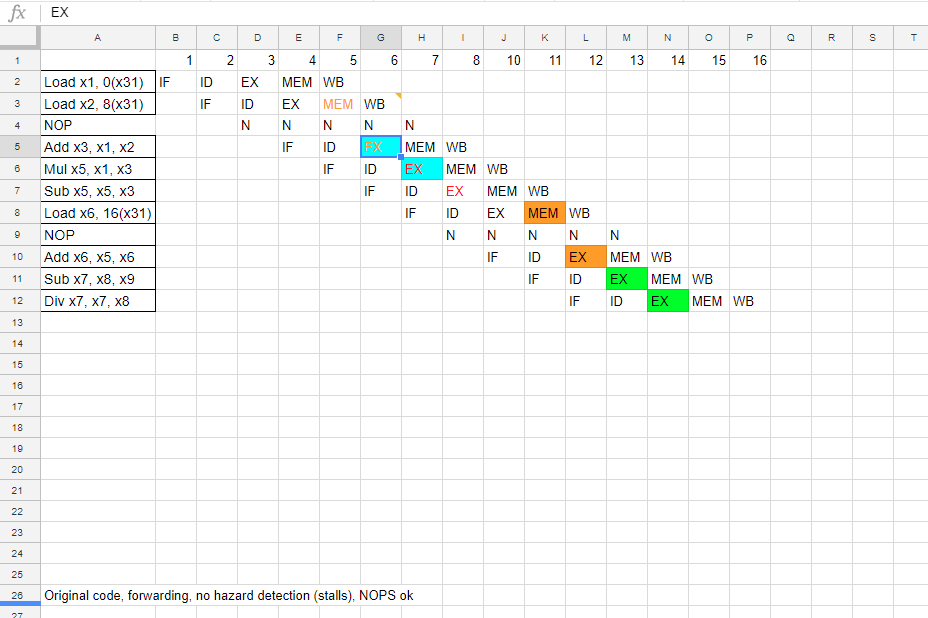


Figure 2: 5-stage pipeline with forwarding and without hazard detection

1. Consider the 5-stage pipeline design **with forwarding**, but without hazard detection logic shown in Figure 2. In this case the compiler is responsible for avoiding RAW hazards. Insert the minimum number of required NOPs into the instruction stream **without re-ordering** instructions to avoid all hazards. Calculate the average CPI of this code, assuming a CPI of one for all instructions. **(3 Points)**

Have the add ins execute align with the writeback of ld2.

* Load executes in data memory.
* You can only forward load after the memory stage.
* check execute memory has priority first since its the latest inst that you have then writeback.
* You delay the decode because by the time you get to execute you need all your variables and registers ready.
  + Example (might not make too much sense without much context):
  + Add x1, x2, x3
  + sub x1, x1, x4
  + mul x4, x1, x3

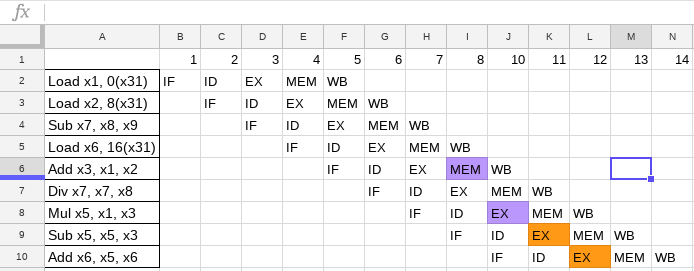


<https://docs.google.com/spreadsheets/d/1K94DBbcZT0YveJwDGUfVhVOzIB_ZbV1EXr_PjV5JDEs/edit?usp=sharing>

|  |
| --- |
| Load x1, 0(x31) |
| Load x2, 8(x31) |
| NOP |
| Add x3, x1, x2 |
| Mul x5, x1, x3 |
| Sub x5, x5, x3 |
| Load x6, 16(x31) |
| NOP |
| Add x6, x5, x6 |
| Sub x7, x8, x9 |
| Div x7, x7, x8 |

CPI of this code: 16 cycles/9 instructions = 1.7777778

1. Consider the 5-stage pipeline design with forwarding, but without hazard detection logic shown in Figure 2. In this case the compiler is responsible for avoiding RAW hazards. Optimize execution time by re-ordering instructions and then insert the minimum number of required NOPs into the instruction stream. Be careful to not change the output of the code when applying transformations. Calculate the average CPI of this code, assuming a CPI of one for all instructions. **(3 Points)**

****

<https://docs.google.com/spreadsheets/d/1K94DBbcZT0YveJwDGUfVhVOzIB_ZbV1EXr_PjV5JDEs/edit?usp=sharing>

|  |
| --- |
| Load x1, 0(x31) |
| Load x2, 8(x31) |
| Sub x7, x8, x9 |
| Load x6, 16(x31) |
| Add x3, x1, x2 |
| Div x7, x7, x8 |
| Mul x5, x1, x3 |
| Sub x5, x5, x3 |
| Add x6, x5, x6 |

CPI of this code: 14 cycles/9 instructions = 1.55555556

1. Consider the 5-stage pipeline design with forwarding, but without hazard detection logic shown in Figure 2. Determine the combinatorial logic required to control the forwarding multiplexers. Use two different expressions, one for forwarding path A and one for forwarding path B. **(2 Point)**

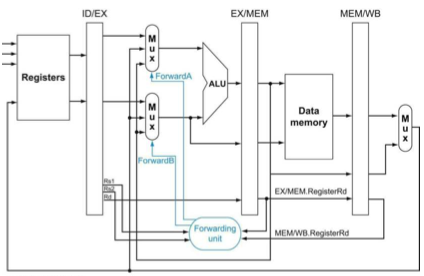


Figure 2: 5-stage pipeline with forwarding and without hazard detection

**Case 1: EX Hazards**

If ((EX/MEM.RegWrite && EX/MEM.RegisterRd != 0)

&&

(EX/MEM.RegisterRd == ID/EX.RegisterRs)):

ForwardA=10

* This chunk of code is saying that if our register number from our execution stage in our earlier command iis the same as the register used as a source/ ALU operand in the following command, and the register being written to and read from are not 0, then forwardA is set to Forward from EX stage.

If ((EX/MEM.RegWrite && EX/MEM.RegisterRd != 0)

&&

(EX/MEM.RegisterRd == ID/EX.RegisterRt)):

ForwardB=10

* Same as above, but for the second ALU operand

**Case 2: MEM Hazards**:

If ((MEM/WB.RegWrite && MEM/WB.RegisterRd != 0)

&&

(MEM/WB.RegisterRd == MEM/WB.RegisterRs)):

ForwardA=01

If ((MEM/WB.RegWrite && MEM/WB.RegisterRd != 0)

&&

(MEM/WB.RegisterRd == MEM/WB.RegisterRt)):

ForwardB=01

* These memory hazards are detected in the same way as the EX hazards above, but instead of checking the current ALU operand registers against the previous ALU results, they are checked against the registers from the MEM/WB stage. In the event that the registers are the same and are not 0 for either operand, the results from MEM/WB are set to be forwarded to the operands via a signal of 01 from either Forward path.

-If both an EX hazard and a MEM hazard are present, do not forward, as a stall or a nop will be necessary.

-This is achieved via an XOR gate for each Forwarding path.

-XOR will set the results to be forwarded if there is either a MEM hazard or an EX hazard for both paths, but not if there is both for either path.

1. You want to minimize the hardware required to implement the forwarding control logic. Which part of the logic terms to produce ForwardA and ForwardB can you share? **(1 Point)**

The overlap in logic will come from both ForwardA and ForwardB using XOR gates to determine if they need to be forwarded. The only gate that we need is an XOR gate for either one, each taking in a 2 bit input, determined by the registers in use in the current and next 2 cycles. If there is a MEM hazard, the input will be 0, 1, EX hazards will give an input of 1, 0. The result of passing these inputs through the XOR gate will then go to the MUX to determine which values are being forwarded.