EEL Hazard Unit Report

Author: Ryan Cramer

Background:

The 32-bit EEL µ-processor in its infancy is essentially a sophisticated copy of the 333 OTTER. Though there are many implementations of the OTTER, there is only one EEL. When it comes to pipelined processor design, hazard detection and branch prediction are two of the biggest challenges. Poorly optimized solutions will result in poorly optimized performance, so much care must be given to selecting a design.

HAZARD (condition): The instruction pipeline contains one or more pending writes to the register file (rd), and decodes an instruction whose source(s) are the same register(s (rs1 or rs2) as the pending instruction(s) destination(s) (rd).

Design Constraints:

1. There are **nine unique opcodes** to implement (without SYS)

OP	**OPCODE**	**RD/RS1/RS2?**
LUI	0110111 = 0x37	RD
AUIPC	0010111 = 0x17	RD
STORE	0100011 = 0x23	RS1 and RS2
LOAD	0000011 = 0x03	RD and RS1
IMM	0010011 = 0x13	RD and RS1
REG	0110011 = 0x33	RD and RS1 and RS2
JAL	1101111 = 0x6f	RD
JALR	1100111 = 0x67	RD and RS1
BRANCH	1100011 = 0x6f	RS1 and RS2

2. There are three stages where data that instructions in Decode could depend on: Execute, Memory, Writeback.

3. Since our DECODER lives in Decode, instructions are not decoded until Decode, meaning that will not worry about handling forwards in the FETCH stage
    a. We can think of instructions as pending until one cycle after Writeback. The destination register will not have an updated value until one cycle after Writeback, due to the synchronous write characteristic of the register file

   b. All instructions besides Branches and Jump (special cases which we will not consider in this document) can be thought of as “popping into existence” in Decode
      i. Therefore, the moment they pop into existence, we should handle them if we can (make implementation easier due to less stages needing forwarding)
      ii. Implements a sort of recursive handling for consecutive hazards (no need to handle hazards past Decode, since they will have already been handled in Decode. That way our HAZARD UNIT won’t have to track state
   c. Loads are the only instruction that will stall then forward to execute if a dependency is behind it in Decode.
      i. Due to the nature of load, register write data is requested in Memory, and available in Writeback.
         1. We forward load-use hazards the Writeback stage, unlike the majority of other instructions, who have their register write data once the ALU generates a result in Execute.
      ii. To handle load use:
         1. Stall the dependency at Decode
         2. Inject a NOP into Execute and move the LOAD to Memory
         3. Allow the dependency to move to Execute and the LOAD to Writeback, where forwarding can occur
         4. Forward from Writeback to Execute (and Decode if needed)

4. Instructions involving loading from memory will only have memory results when the LOAD instruction is in writeback (We directly wire the output of the DMEM in the Memory stage to the RF_MUX in Writeback, instead of buffering it with the MW (Memory/Writeback) pipeline register.
   - Data Memory’s RD_EN signal is asserted when the LOAD is in Memory, and DATA_OUT is available in the next cycle when the LOAD is in Writeback.
   - Therefore, any dependency directly trailing the LOAD in the pipeline must wait in Decode while a NOP is artificially fed into Execute, putting the necessary distance between the LOAD and its dependency to be able to forward the LOAD to the ALU while the dependency is in Execute.

5. Instructions involving control transfer must be handled carefully.
   - JAL - pipeline must be flushed, PC $\leftarrow$ PC+imm, x[rd] $\leftarrow$ PC+4
   - JALR - pipeline must be flushed, PC $\leftarrow$ (rs1 + imm&~1), x[rd] $\leftarrow$ PC+4
   - BRANCH - pipeline may be flushed, wait until execute,

6. Hazards must resolve forwards and stalls, which only happen after Decode for non control-transfer instructions, while the Branch Predictor will handle control-transfer from Fetch to Execute, depending on information availability.

7. Depending on the impact to critical path, clock speed, and routing congestion, we will weight the options before us, test them, and decide which implementation is the most optimal to have the greatest performance.We will select a design after considering:

    1. IPC (instructions per cycle)
    2. CPI (cycles per instruction)
    3. Clock Frequency
    4. Estimated Power

Design Paradigms:

1. Sequential Solution (Finite State Machine)
    - Implements algorithmic step-by-step solution of problem
    - complicated to handle using states when there are multiple dependencies to forward/handle.
        - n = number of opcodes
            - must add unique cases
            - Ex: sw uses rs1 and rs2, however it needs rs1 in Execute and rs2 in Memory, meaning for rs1 you must stall and rs2 you can forward or must wait to forward if rs2 is the rd of a load operation (it gets complicated quickly, and all the while other instructions are pending in the pipeline)
        - k = 4 (What instruction is in Decode, Execute, Memory, and Writeback?)
2. Combinational Solution (Gate-Level Logic)
    - Not able to do sequences (stall then resolve sequence)
        - This means you must know what the instruction and data is in each cycle using wires, because you cannot track a state across the pipeline
    - Can be designed to handle all hazards in one cycle
    - Logic can be reduced and not require the use of registers

Because our target is an ASIC using older 130nm technology, prototyping with a modern FPGA such as the Basys 3 is not feasible, as such devices use modern transistor sizing (<50nm). We must instead run our RTL through tools that transform the RTL into a technology mapped gate-level netlist, which is then passed to PnR (place and route) tools that check if our design is electrically feasible.

** Authors Note: Sky130's flip-flops will be the number one source of headache when trying to reach timing closure. Though our critical path may be greatly impacted, it would be wise to implement large modules as combinational units, to ensure that the design passes Static Timing Analysis.**

Data Hazards (No Jump/Branch Handling)

There are primarily 9 hazard types that don't involve control flow (updating the PC):

NON-SPECIAL (any instruction besides control transfer could use these)

I. Forward RS1 (Execute's RD Data) from Execute to Decode (Decode's RS1)
II. Forward RS1 (Memory's RD Data) from Memory to Decode (Decode's RS1)
III. Forward RS1 (Writeback's RD Data) from Writeback to Decode (Decode's RS1)

IV. Forward RS2 (Execute's RD Data) from Execute to Decode (Decode's RS2)
V. Forward RS2 (Memory's RD Data) from Memory to Decode (Decode's RS2)
VI. Forward RS2 (Writeback's RD Data) from Writeback to Decode (Decode's RS2)

SPECIAL

Post STALL Handling:

VII. Forward RS1 (Writeback's RD) from Writeback to Execute
VIII. Forward RS2 from Writeback to Execute

LOAD→MEM Skip Stall (RS2)

IX. Forward RS2 from Writeback to Memory (unavailable for RS1)

* RS2 is not needed by STORE instructions in Execute (STORE Memory[dest] is formed using RS1, not RS2), so you do not need to waste a clock cycle to stalling. You can instead skip the stall, and simply forward the result of the LOAD directly from Writeback to the STORE's RS2 in Memory (Used for DMEM Data_In)*

Flow Hazards (Jump/Branch Handling)

You have 5 feasible options for handling flow hazards:

1. Implement Fetch, Decode, Execute handling for JUMP & BRANCH **(most hardware)**

2. Implement Decode, Execute handling for JUMP & BRANCH **(decode critical path ++)**

3. Implement Fetch, Decode handling for JUMP; Decode, Execute for BRANCH **(no BP)**

4. Implement Fetch, Decode handling for JUMP, Execute for BRANCH **(low hardware, no BP)**

5. Implement Decode, Execute handling for JUMP, Execute for BRANCH **(least hardware)**

Fetch, Execute, Decode Handling of Branches and Jumps (predictor required)

There are 8 extra control transfer hazards to watch out for:

I. Forward RS1 (Decode's RS1) from Decode to Fetch
II. Forward RS1 (Execute's RD or RS1) from Execute to Fetch
III. Forward RS1 (Memory's RD or RS1) from Memory to Fetch
IV. Forward RS1 (Writeback's RD or RS1) from Writeback to Fetch

V. Forward RS2 (Decode's RS2) from Decode to Fetch
VI. Forward RS2 (Execute's RD or RS2) from Execute to Fetch
VII. Forward RS2 (Memory's RD or RS2) from Memory to Fetch
VIII. Forward RS2 (Writeback's RD or RS2) from Writeback to Fetch

JAL:

- Fetch Handling: (no sources)
  - handle in Fetch, unless pending control transfer in D or E)
- Decode Handling: (no sources)
  - only handled here if Fetch was blocked by a pending control transfer instruction)

JALR:

- Fetch Handling: Requires RS1
  - Hazards
    - Forward if RS1 is not a destination in the pipeline and is a source elsewhere in the pipeline (RS1 -> RS1)
    - Forward if RS1 has a pending write:
      - Execute (no LOAD)
      - Memory (no LOAD)
      - Writeback
    - If RS1 is Decode WADDR, wait until Decode to resolve
    - If RS1 not seen in pipeline, wait until Decode
- Decode Handling: Requires RS1
  - Get RS1 from Register File
  - Forward if RS1 has a pending write
    - Execute (no LOAD)
    - Memory (no LOAD)
    - Writeback
  - If load-use, you must stall then jalr in Execute
  - Flush Fetch

        - Don't take if unresolved branch in Execute

- Execute Handling: Requires RS1 from a LOAD
    - Only happens after a load-use stall
    - Flush Fetch and Decode

BRANCH:

Uses Dynamic Predictor FSM to track likelihood of taking the branch

- Implement 3 (or possibly 4) states to track branch likelihood

    - TAKE
    - TOSSUP
    - DON'T TAKE

    or

    - LOW
    - MEDIUM-LOW
    - MEDIUM-HIGH
    - HIGH
- Implement a branch_file that holds a finite number of branch outcomes/pending states [less than or equal to 32 bits]

- FETCH Handling (options): Requires RS1 and RS2 to be forwarded
    1. Get RS1 or RS2 from the pipeline, compare, take branch
    2. Take branch because Predictor Likelihood at MAX
    3. Don't take branch because unresolved JAL or JALR in pipeline
- DECODE Handling (options): Requires RS1 and RS2
    - Get RS1 or RS2 from the register, compare them and branch
    - Forward RS1 and/or RS2, compare them and branch
    - Take branch because RS1 and/or RS2 not available, but predictor at MEDIUM-HIGH
    - Don't take branch because RS1 and/or RS2 not available, but predictor at MEDIUM-LOW
- EXECUTE HANDLING: Always handle here
    - Writeback's LOAD hazard forwards its write data, which will help us determine whether or not we take the branch
    - Take branch or don't…you'll know by this stage