CSE 30321 – Lecture 12 – Datapath Expansion Example

Adding a New Instruction:

Referring to the extra handouts, modify the multi-cycle datapath shown below to support a new instruction: load++ \$x n(\$y).

The RTL for load ++ is as follows:

$x ← Mem(n + RF($y))
$y ← $y + 4

Show any necessary changes to the FSM as well.

Part A:

Describe – cycle-by-cycle (starting with Fetch) – what this instruction needs to do:

Item 1: Recap of LW + new functionality.

Remember what the base load does…

- lw <destination register>, offset(<register with value used to calculate address sent to memory>)
- Address sent to memory = offset + data in <register value used to calculated address…>
- Data in destination register = data at address calculated above

Now: ALSO, add 4 to <register with value used to calculate address…>

Item 2: Review RTL, discuss different options for solving problem.

- Fetch:
  - IR ← Mem(PC)
  - PC ← PC +4

  } Same for every instruction
- Decode:
  - A ← RF(IR[25:21])
  - B ← RF(IR[20:16])
  - ALUOut ← PC + (sign-extend (IR[15:0] << 2)

  } Same for every instruction

Item 2a: Digression – have assumed BEQ = 3 CCs (see Lecture 09 slide)

- Why is ALUOut line here…
- For BEQ to be done in 3 CCs, need to calculate address in CC #2
- Otherwise, would do comparison and address calculation in CC #3 and would need antoher ALU
- By doing this in CC #2, we can re-use the ALU … and it does no harm
- Strategy: do something ASAP if HW is available and idle

Item 2b: Back to lw++

- Execute:
  - ALUOut = A + sign-extend(IR[15:0])
- Memory:
  - MDR ← Mem[ALUOut]
  -
  - ***Question: Can we update the \$y register here?***
    - Yes!
    - The ALU is idle for all intent and purposes
    - Therefore, can also do: \$y ← \$y + 4

- Write Back:
  - RF(IR[20:16]) ← MDR # Just do normal lw first (i.e. write data back to register)
  - ***Question: Can we also update $y here? Actually, there are 2 answers…***
    - Option A: Yes… but we need to add more HW…
      - (Namely another path to write data to the register file is needed…)
      - **See datapath on overheard; we'll draw in Part B.**
    - Option B: No… we need to add another state…
      - Look at mux … you can only select the MDR or ALUOut … not both!
      - Begs another question… ***how do you modify the FSM?***
        - Add state coming off of State 4. This would allow us to handle the load++ case. There would then be a path back to fetch

Part B:
Given your answer to Part A, how would you modify the datapath or state diagram?

If Option A, we would need to modify both the state machine and the datapath.

One solution would involve:

Datapath Changes:
- Change "Write Register" on the RF to Write Register 1 – and add Write Register 2
- Similarly, change "Write Data" on the RF to "Write Data 1" – and add "Write Data 2"
- **See changes on datapath handout.**
    - Need to write \$y and \$x simultaneously, therefore need register address and value.
    - Value will come from Instruction[25:21]
        - I-Type: opcode, rs, rt, immediate
        - For lw: reg[rs] ← Memory(reg[rt] + immediate)

State Diagram Changes:
- This will require the addition on 1 control signal – RegWrite(2)
    - Realistically, would need to add to other states too
        - (b/c you don't want to write 2 values for other instructions)
- A new state would still need to be added however…
    - State 12 would branch off of State 3 to ensure that 2 registers are written, not just 1.
    - **See changes on FSM handout.**

If Option B, we would need to add another state as specified above.
- However, no new control signals would need to be added.

- Let's assume we would write data from the MDR first – this is just the previous State 4.
- Then in State 13, we would want to write data in ALUOut to the register file.
- Therefore, we would need to do the following…
    - In State 3, we would need to add the following:
        - ALU control = ADD
        - ALUSrcB = 01 (select #4)
        - ALUSrcA = 1 (select A register)
    - For State 13 (added above) we would need to:
        - Add path from IR[25:21] to multiplexor
            - RegDst now becomes a 2 bit control signal
        - MemToReg = 0 (select ALUOut)
        - RegWrite = 1 (enable register to be written)
    - **Do this in FSM diagram after we write on board…**

Some general comments:
- These are the kind of tradeoffs that you might consider. Do you add more HW or do you pay an extra CC?
- If we did this with and Add and a LW, how much savings would there be (assuming multi-cycle)?
    - 5 + 4 = 9 CCs vs. 6CCs
    - 50% faster to replaced 2 instruction sequence with new instruction
    - How often could you use it?

Setting Up Pipelining:

If we take the above approach, how should we break up the instructions?

Think about each step … what needs to be done to ensure R, I, and J type instructions all work?

- *Hint: Think about each instruction and think about what can be done in parallel.*

(1) IF	**(2)** ID	**(3)** EX	**(4)** M / R-Type	**(5)** WB
(every) Fetch instruction at address in PC	**(every)** Decode instruction type (i.e. send opcode to control logic)	**(lw/sw)** Compute memory address	**(lw)** Read memory, store result in MDR (memory-data-reg)	**(lw)** Write MDR to register file.
(every) Store result in instruction register (IR)	**(every)** Store results read from the register file	**(R-type)** Perform correct logical operation	**(R-type)** Write ALUout to reference	
(every) increment PC	**(jump)** Compute new PC address****	**(Branch)** Update PC post comparison	**(sw)** Write to memory	
		(All) Store intermediate result in ALU **(Note)** All use ALU	Therefore, do memory reference or register write…	

There is a catch with the pipelining approach:

- Each instruction must go through every stage
- Therefore, each instruction will take 5 CCs, and an ALU instruction will needlessly spend a CC in the memory stage
- However, if an instruction finishes each CC – so an ALU now takes 1 CC instead of 4 CCs – the speedup is still 4X!