CS4617 Computer Architecture

Lecture 9: Pipelining

Reference: Appendix C, Hennessy & Patterson

Dr J Vaughan

13 October 2014

5-stage pipeline

	Clock cycle								
Instr No	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
i	IF	ID	EX	MEM	WB				
i+1		IF	ID	EX	MEM	WB			
i+2			IF	ID	EX	MEM	WB		
i+3				IF	ID	EX	MEM	WB	
i+4					IF	ID	EX	MEM	WB

Table: Figure C.1: Simple RISC pipeline

Forwarding

- Forwarding can be generalised to pass a result directly to the FU that requires it
- A result is forwarded from the pipeline register at the output of one unity to the input of another, rather than from the result of a unit to the input of the same unit
- Example
  - DADD R1, R2, R3
  - LD R4, 0(R1)
  - SD R4, 12(R1)
- To prevent a stall in this sequence, need to forward the values of ALU output and memory unit output from the pipeline registers to the ALU and data memory inputs
- Figure C8 shows all the forwarding paths for this example.

Forwarding of operand required by stores during MEM

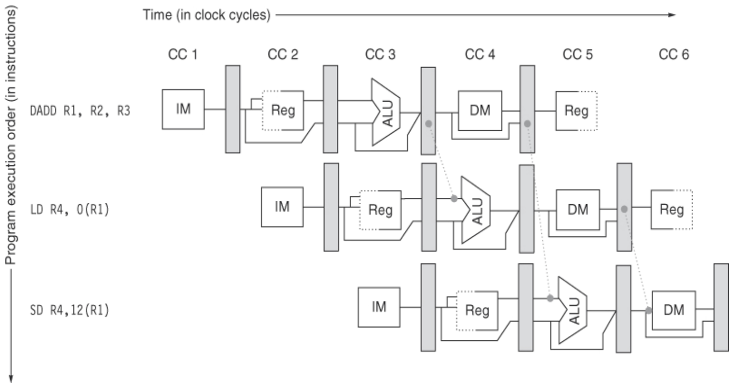

Figure C.8 Forwarding of operand required by stores during MEM. The result of the load is forwarded from the memory output to the memory input to be stored. In addition, the ALU output is forwarded to the ALU input for the address calculation of both the load and the store (this is no different than forwarding to another ALU operation). If the store depended on an immediately preceding ALU operation (not shown above), the result would need to be forwarded to prevent a stall.

Data Hazards requiring stalls

- Not all data hazards can be handled by bypassing
- Example
  - LD R1, 0(R2)
  - DSUB R4, R1, R5
  - AND R6, R1, R7
  - OR R8, R1, R9
- See Figure C9

A data hazard that cannot be handled by bypassing

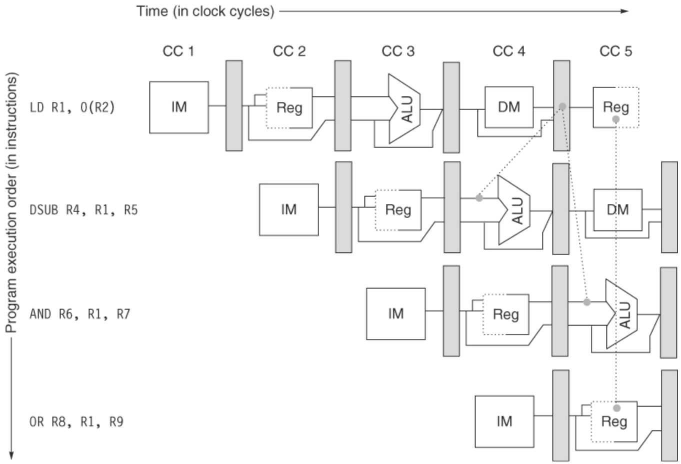

Figure C.9 The load instruction can bypass its results to the AND and OR instructions, but not to the DSUB, since that would mean forwarding the result in "negative time."

Data Hazards requiring stalls (continued)

- LD does not have the data until the end of clock cycle 4 (its MEM cycle)
- DSUB needs to have the data by the beginning of clock cycle 4 $\Longrightarrow$ the data hazard from the result of a load instruction cannot be completely eliminated with simple hardware
- The result can be forwarded immediately to the ALU for use in the AND operation that begins 2 clock cycles after the load.
- The OR instruction receives the value through the register file
- DSUB receives the forwarded result 1 clock cycle too late
- Hardware called a *pipeline interlock* must be added to detect a hazard and stall the pipeline until it is cleared.
- CPI for the stalled instruction increases by the length of the stall (1 clock cycle in this case)
- Figure C10 shows the pipeline before and after the stall
- Forwarding to AND goes through the register file because the stall causes instructions starting with DSUB to move 1 cycle later

Data Hazards requiring stalls (continued)

	Clock cycle							
Instr No	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
LD R1, 0(R2)	IF	ID	EX	MEM	WB			
DSUB R4, R1, R5		IF	ID	EX	MEM	WB		
AND R6, R1, R7			IF	ID	EX	MEM	WB	
OR R8, R1, R9				IF	ID	EX	MEM	WB

Table: Figure C10(a): The MEM cycle of the load produces a value that is needed in the EX cycle of the DSUB, which occurs at the same time

Data Hazards requiring stalls (continued)

	Clock cycle								
Instr No	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
LD R1, 0(R2)	IF	ID	EX	MEM	WB				
DSUB R4, R1, R5		IF	ID	stall	EX	MEM	WB		
AND R6, R1, R7			IF	stall	ID	EX	MEM	WB	
OR R8, R1, R9				stall	IF	ID	EX	MEM	WB

Table: Figure C10(b): Resolution of the problem with a stall

Branch Hazards

- When a branch is executed, it may change PC to PC+4 or something different
- If I_i is a taken branch, PC is not normally changed until the end of ID, after completion of address calculation and comparison
- Figure C11 shows that the simplest way to deal with branches is to redo instruction fetch after the branch, having detected the branch during ID
- The first IF is not useful and amounts to a stall
- If the branch is untaken, repetition of the IF is not necessary since the correct instruction has been fetched
- One stall cycle for every branch gives a performance loss of 10% to 30% so techniques to deal with this are needed

Branch causing 1-cycle stall

	Clock cycle								
Instr No	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Branch instruction	IF	ID	EX	MEM	WB				
Branch successor		IF	IF	ID	EX	MEM	WB		
Branch successor + 1				IF	ID	EX	MEM	WB	
Branch successor + 2					IF	ID	EX	MEM	WB

Table: Figure C11: A branch causes a 1-cycle stall in the 5-stage pipeline

Reducing Pipeline Branch Penalties

- 4 compile time schemes
- The actions for a branch are static: fixed for each branch during the entire execution

1. Freeze or flush the pipeline: the simplest scheme
    - Hold or delete any instructions after the branch until the branch destination is known
    - This is the solution shown in Figure C11
    - The branch penalty is fixed and cannot be reduced by software

Reducing Pipeline Branch Penalties

2. Treat every branch as not taken.
    - Do not change processor state until branch outcome is definitely known
    - Complexity: when might state change? How to reverse a change?
    - This is known as predicted-not-taken
    - Continue to fetch instructions as if there were no branch
    - Restart fetch at target address (and turn previously fetched instruction into a NOP) if the branch is taken
    - See Figure C12

Predicted-not-taken when the branch is untaken

	Clock cycle								
Instr No	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

Table: Figure C12(a) Predicted-not-taken when the branch is untaken

Predicted-not-taken when the branch is taken

	Clock cycle								
Instr No	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction i+1		IF	**idle**	**idle**	**idle**	**idle**			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

Table: Figure C12(b) Predicted-not-taken when the branch is taken

Reducing Pipeline Branch Penalties

3. Treat every branch as taken
   - Fetch and execute from target address as soon as it has been computed
   - In this processor, the target address is not known earlier the branch outcome, so there is no advantage in this scheme
   - In processors where the target address is known before the branch outcome, this may make sense
   - The compiler can improve performance by matching code to the hardwares use of predicted-taken or predicted-not-taken

Reducing Pipeline Branch Penalties

4. Delayed branch
   - Execution cycle with a branch delay of 1
   - Branch instruction
     Sequential successor 1
     Branch target if taken
   - The sequential successor is in the *branch delay slot*
     This is executed in either case
   - Most processors have a 1-instruction branch delay
   - See Figure C13

Delayed branch when branch is untaken

	Clock cycle								
Instr No	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i+1)		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

Table: Figure C13(a) Behaviour of a delayed branch if branch is untaken

Delayed branch when branch is untaken

	Clock cycle								
Instr No	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i+1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Table: Figure C13(b) Delayed branch behaviour is the same if branch is taken

Delayed branch (continued)

- Note that another branch should not be put in the delay slot
- The compiler inserts useful instructions in the delay slot
- See Figure C14 for optimisations

Forwarding of operand required by stores during MEM

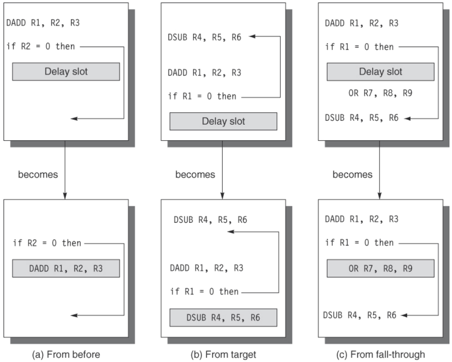

Figure C.14 Scheduling the branch delay slot. The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of R1 in the branch condition prevents the DADD instruction (whose destination is R1) from being moved after the branch. In (b), the branch delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall-through as in (c). To make this optimization legal for (b) or (c), it must be OK to execute the moved instruction when the branch goes in the unexpected direction. By OK we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, in (c) if R7 were an unused temporary register when the branch goes in the unexpected direction.

Limitations on delayed-branch scheduling

1. Restrictions on instructions that may be scheduled into delay slots
2. Ability to predict at compile time whether a branch is taken or not

Performance of branch schemes

- Assume ideal CPI = 1
- $Pipeline\ speedup = \frac{Pipeline\ \ depth}{1+Pipeline\ \ stall\ \ cycles\ \ from\ \ branches}$
- $Pipeline\ stall\ cycles\ from\ branches =$
  $Branch\ frequency \times Branch\ penalty$
- $Pipeline\ speedup = \frac{Pipeline\ \ depth}{1+Branch\ \ frequency \times Branch\ \ penalty}$
- Example
  In deeper pipelines (e.g., MIPS R4000) it takes at least 3 pipeline stages before the branch target address is known and an additional cycle before the branch condition is evaluated, assuming no stalls in the conditional comparison

Performance of branch schemes: Example

- Branch penalties for the 3 simplest prediction schemes are shown in Figure C15
- Find the effective CPI arising from branches for this pipeline, assuming the following frequencies
  - Unconditional branch: 4%
    Conditional branch, untaken: 6%
    Conditional branch, taken : 10%
- CPI calculated by multiplying the relative frequencies by the corresponding penalties: see Figure C16
- Note that the differences between schemes are increased with longer delay

Branch penalties for prediction schemes

Branch scheme	**Penalty unconditional**	**Penalty untaken**	**Penalty taken**
Flush pipeline	2	3	3
Predicted taken	2	3	2
Predicted untaken	2	0	3

Table: Figure C15: Branch penalties for prediction schemes

CPI penalties for three branch prediction schemes and a deeper pipeline

Branch scheme	**Unconditional branches**	**Untaken conditional**	**Taken conditional**	**All branches**
Frequency of event	4%	6%	10%	20%
Stall pipeline	0.08	0.18	0.3	0.56
Predicted taken	0.08	0.18	0.2	0.46
Predicted untaken	0.08	0.00	0.3	0.38

Table: Figure 16: CPI penalties for three branch prediction schemes and a deeper pipeline

Reducing the cost of branches: Prediction

- Static prediction: uses information available at compile time
- Dynamic prediction: based on program behaviour

Static branch prediction

- Can improve compile-time prediction based on information from earlier runs
- An individual branch is often biased towards taken or untaken
- Effectiveness of any branch prediction scheme depends on the accuracy of the scheme and the frequency of conditional branches (3% to 24% in SPEC)
- The mis-prediction rate for integer programs is higher and these usually have a higher branch frequency
- This limits the usefulness of static branch prediction

Dynamic branch prediction

- Simple scheme: Branch-prediction buffer
- A small memory indexed by the lower part of the address of the branch instruction
- 1 bit to indicate branch taken/not taken
- Simple buffer, no tags
- Reduces branch delay when it is longer than the time to compute the possible target PC
- Do not know if prediction is correct because many branches have some low-order address bits.
- Prediction is a hint assumed correct, fetching begins in predicted direction. If hint is wrong, prediction bit is inverted and stored.
- 1-bit prediction: assume branch almost always taken
- If branch not taken, this is a misprediction, bit changed next time, branch prediction = not taken, but branch is taken, a misprediction
- So incorrect prediction occurs twice on the less-taken path
- A 2-bit prediction scheme must miss twice before it is changed: see Figure C18

Forwarding of operand required by stores during MEM

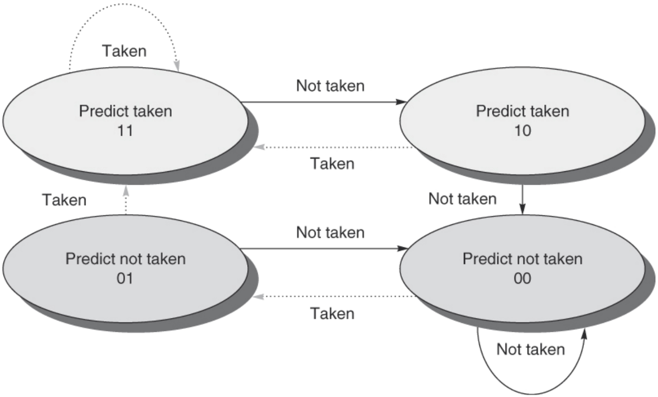

Figure C.18 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually a specialization of a more general scheme that has an *n*-bit saturating counter for each entry in the prediction buffer. With an *n*-bit counter, the counter can take on values between 0 and 2n – 1: When the counter is greater than or equal to one-half of its maximum value (2n – 1), the branch is predicted as taken; otherwise, it is predicted as untaken. Studies of *n*-bit predictors have shown that the 2-bit predictors do almost as well, thus most systems rely on 2-bit branch predictors rather than the more general *n*-bit predictors.

Simple implementation of unpipelined MIPS

- Integer subset of MIPS
- Load/store word
- Branch if equal to zero
- Integer ALU operations
- Less aggressive branch implementation initially
- MIPS instruction implemented in at most 5 clock cycles

Implementation of unpipelined MIPS

1. Instruction fetch cycle (IF)
    - IF $\leftarrow$ Mem[PC];
      NPC $\leftarrow$ PC+4;
    - Operation: IR used to hold instruction that is needed on subsequent clock cycles
      NPC holds next sequential PC

Implementation of unpipelined MIPS

2. Instruction decode/register fetch cycle (ID)
   - A $\leftarrow$ Regs[Rs];
     B $\leftarrow$ Regs[Rt];
     Imm $\leftarrow$ sign-extended immediate field of IR
   - Operation: Decode instruction and read registers Rs and Rt from the register file. Put them in temporary registers (A,B) for use in later clock cycles.
     Sign-extend lower 16 bits of IR and put in temporary register Imm
     Decoding in parallel with register read possible because of fixed-field encoding in MIPS
     Sign-extend of immediate is also possible at the same time due to fixed-field encoding

Implementation of unpipelined MIPS

3. Execution/effective address cycle (EX)
   ALU performs 1 of 4 functions, depending on MIPS instruction type
   - Memory reference
     ALUOutput $\leftarrow$ A + Imm;
     Operation: effective address formed and put in register ALUOutput
   - Register-register ALU instruction ALUOutput $\leftarrow$ A *func* B;
     Operation: Binary function *func* performed on A, B and result put in register ALUOutput
   - Register-immediate ALU instruction ALUOutput $\leftarrow$ A *op* Imm;

Implementation of unpipelined MIPS

3. Execution/effective address cycle (EX) (continued)
   - Branch
     ALUOutput $\leftarrow$ NPC + (Imm << 2);
     Cond $\leftarrow$ (A == 0);
     Operation: The sign-extended immediate value in Imm is multiplied by 4 to give a word offset and then added to NPC to give the branch target address
     Register A is checked to see if branch is taken
     Only one branch is being considered, BEQZ, so comparison is with 0
     Note BEQZ is a pseudo-instruction that translates to BEQ with R0 as an operand
     The load-store architecture of MIPS allows effective address and execution cycles to be combined in a single clock cycle
     Instructions not included are various jumps, similar to branches

Implementation of unpipelined MIPS

4. Memory access/branch completion cycle (MEM)
   - $\forall$ $instructions : PC \leftarrow NPC;$
   - Memory reference
     LMD $\leftarrow$ Mem[ALUOutput] or Mem[ALUOutput] $\leftarrow$ B;
     - Operation: Access memory if needed. If instruction is a load, put data from memory in LMD (load memory data) register. If instruction is a store, write data from the B register into memory
   - Branch
     If (cond) PC $\leftarrow$ ALUOutput
     - Operation: If the instruction branches, use the target address in ALUOutput

Implementation of unpipelined MIPS

5. Write-back cycle (WB)
    - Register-register ALU instruction
      Regs[Rd] $\leftarrow$ ALUOutput;
    - Register-immediate ALU instruction
      Regs[Rt] $\leftarrow$ ALUOutput;
    - Load instruction
      Regs[Rt] $\leftarrow$ LMD;
    - Operation: Write the result to the register file
      If instruction is load, source = LMD, dest = Rt
      If instruction is ALU, source = ALUOutput

   Figure C21 shows the data path

MIPS data path implementation (unpipelined)

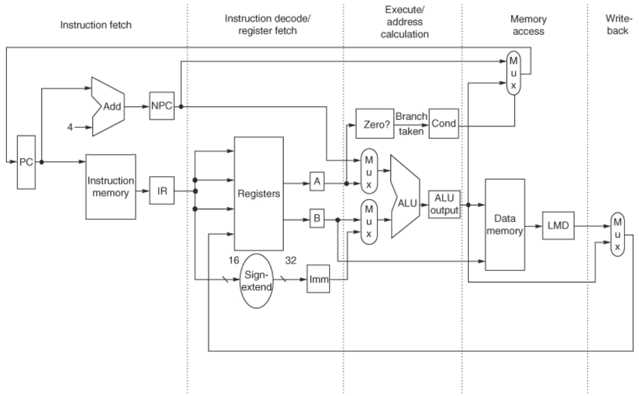

Figure C.21 The implementation of the MIPS data path allows every instruction to be executed in 4 or 5 clock cycles. Although the PC is shown in the portion of the data path that is used in instruction fetch and the registers are shown in the portion of the data path that is used in instruction decode/register fetch, both of these functional units are read as well as written by an instruction. Although we show these functional units in the cycle corresponding to where they are read, the PC is written during the memory access clock cycle and the registers are written during the write-back clock cycle. In both cases, the writes in later pipe stages are indicated by the multiplexer output (in memory access or write-back), which carries a value back to the PC or registers. These backward-flowing signals introduce much of the complexity of pipelining, since they indicate the possibility of hazards.

Implementation of unpipelined MIPS

- At the end of each clock cycle, every value computed during that clock cycle and needed on a later clock cycle (in this instruction or the next) is written into a storage device
- The temporary registers hold values between clock cycles for 1 instruction
- Other storage elements hold values between successive instructions
- This multi-cycle implementation approximates early processor designs
- Microcode control could be used to give a more complex processor
- Hardware redundancies could be removed but are left here to give a base for a pipelined implementation
    - For example, there are 2 ALUs: one to increment the PC and one for ea and ALU computation. Since they are not needed on the same clock cycle, they could be merged by using extra multiplexers.
    - It is also possible in this unpipelined processor to store data and instructions in the same memory.