CS147 - Lecture 15

Kaushik Patra
(kaushik.patra@sjsu.edu)

1

- Pipeline Data Hazard
- Pipeline Control Hazard
- Pipeline Implementation

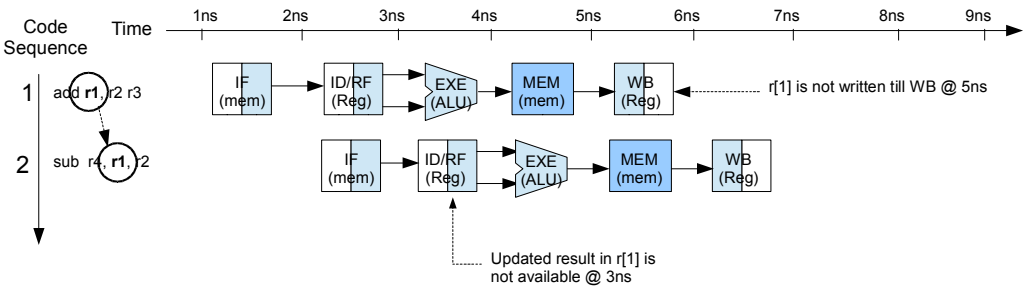
Reference Books:

1) Chapter 6 of 'Computer Organization & Design' by Patterson & Hennessy

Pipelining Data Hazard ...

2

Data Hazard

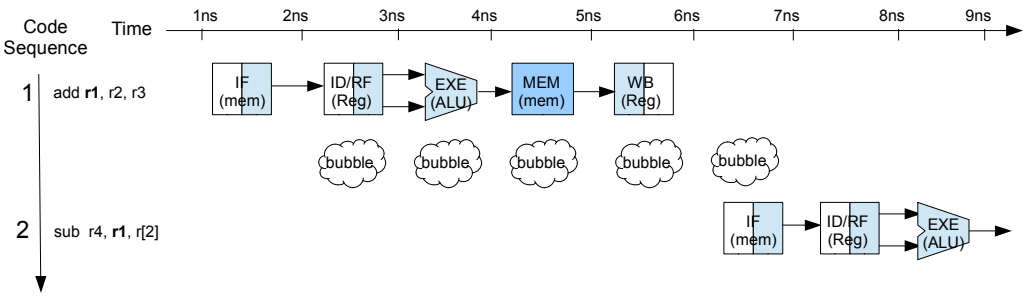


- Result of instruction 1 (r1) is used in instruction 2.

3

- Instruction 2 is depending on instruction 1 since it is depending on result from instruction 1.
- Result of an instruction is usually available after WB stage of the execution. Therefore at the ID/RF stage of instruction 2 (@3ns), the result from instruction 1 can not be available. The result of instruction 1 is available into the register file only after 5ns in this example. As a result, instruction 2 must be held till 6ns before it can be executed. This is known as data hazard, where needed argument value of certain instruction is still unavailable to use.

Data Hazard Solution - Stalling

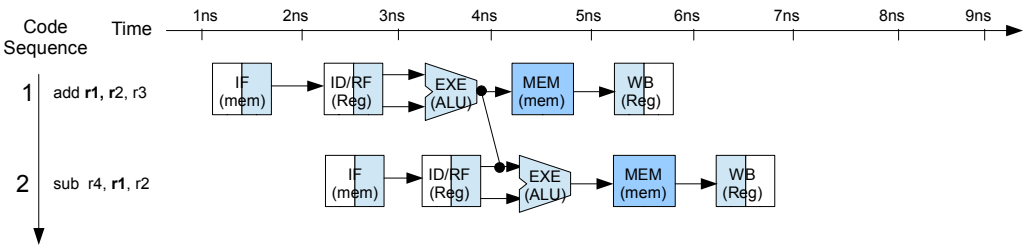


- Stall the pipeline till the data is ready – in this case it is 5 cycle stall of the pipeline.

4

- The obvious solution to resolve such data hazard is by stalling the pipeline. If we stop the pipeline for 5 cycles, after instruction 1 is fetched, to give time to instruction 1 to be completed, we can avoid data hazard present for instruction 2. However, if we make this stalling a standard practice, we'll loose most of the performance gain from pipeline architecture, because we are making instruction execution equivalent to non-pipelined architecture. Our target program is full of such data dependencies between instructions – hence it is worth to think of alternative to resolve such data hazard.

Data Hazard Solution - Forwarding

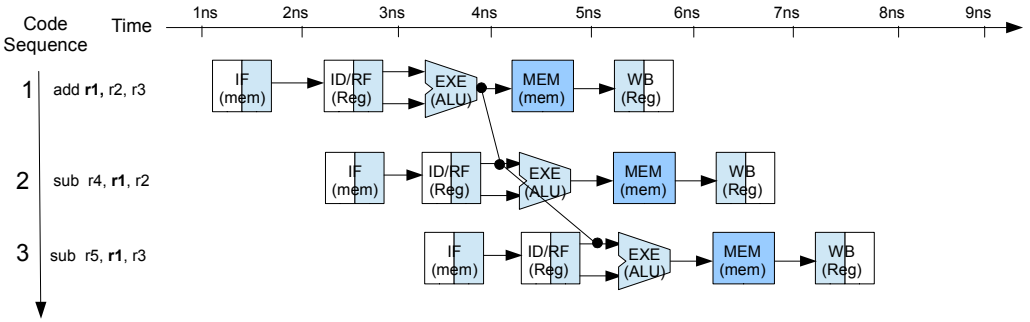


- Use the result directly from EXE stage of previous instruction.

5

- One observation for this example is that the data needed for EXE stage of instruction 2 is already available at EXE stage of instruction 1. If we can use this EXE stage result from ALU for instruction 1 into the EXE stage of instruction 2, we do not need to wait till the result is written back into the register file. Such technique of using uncommitted result into forwarding stage of next instructions in the pipeline is known as data forwarding.

Data Hazard Solution - Forwarding

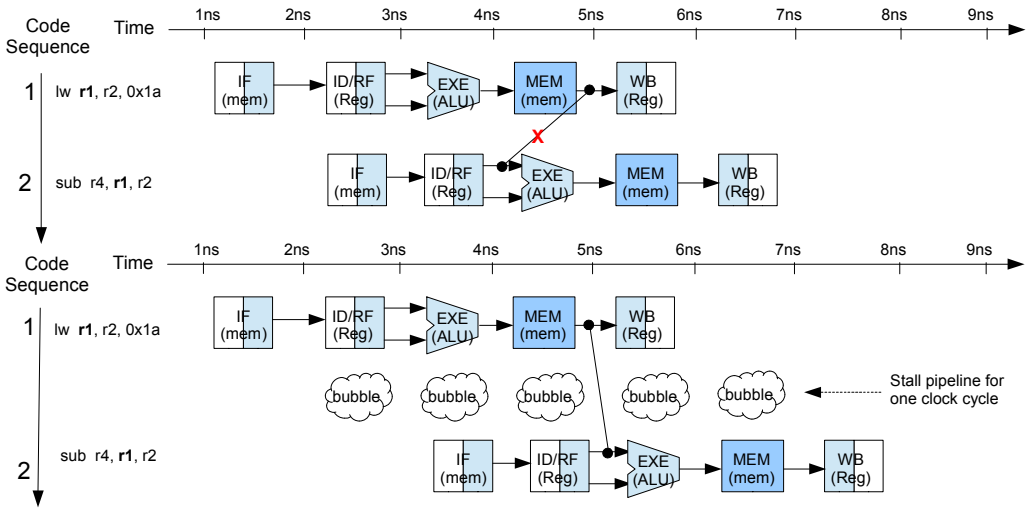


- Data forwarding can happen if register fetch happens for a dependent instruction at the same time or later of the EXE / MEM (for 'lw' as previous instruction) stage of the instruction upon which the current instruction is depending on.

6

- Pipeline architecture allows us to use uncommitted results of an instruction into temporally advance stages of successive instructions. In this example, uncommitted result from ALU at the EXE stage of instruction 1 is used for both instruction 2 and 3, which are data dependent on instruction 1.

Stalling & Forwarding

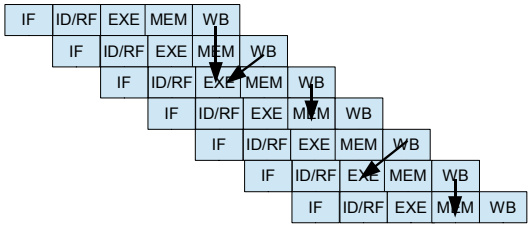


- In some cases, stalling can not be avoided – target is to minimize number of stalled cycle as much as possible. 7

- However, in some cases, we can not avoid stalling the pipeline since data forwarding technique works only if the uncommitted result is used in temporally advanced stage of successive instruction. In this example, uncommitted result of instruction 1 is available after the MEM stage is done for instruction 1. Since instruction 2 needs this result at its EXE stage, which is temporally behind the MEM stage of instruction 1, it can not be used. In this situation stalling may be required.

Data Hazard Solution - Reordering

1	r1 ← M[r0+0x0];	lw r1, r0, 0x0
2	r2 ← M[r0+0x4];	lw r2, r0, 0x4
3	r3 ← r1 + r2;	add r3, r1, r2
4	M[r0+0x12] ← r3;	sw r3, r0, 0x12
5	r4 ← M[r0+0x8];	lw r4, r0, 0x8
6	r5 ← r1 + r4;	add r5, r1, r4
7	M[r0+0x16] ← r5;	sw r5, r0, 0x16



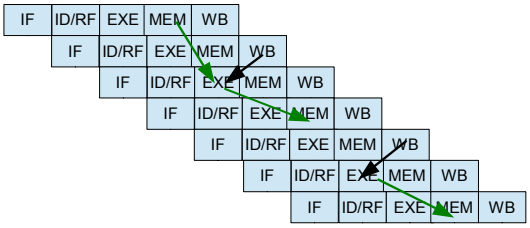
- Instruction 3, 4, 6, 7 has data hazard
  - 3 depends on 1, 2
  - 4 depends on 3
  - 6 depends on 5
  - 7 depends on 6

8

- There is another technique to avoid stalling, as in the example in slide 7. This technique is instruction re-ordering, which is done by compiler. In this given example, data hazards are identified with arrows. For example, EXE stage of instruction 3 is depending on WB stage of instruction 1 and 2. Similarly, MEM stage of instruction 4 is depending on WB of instruction 3. In the similar notion, EXE of 6 depends on WB of 5 and MEM of 7 depends on WB of 6.

Data Hazard Solution - Reordering

1	r1 ← M[r0+0x0];	lw r1, r0, 0x0
2	r2 ← M[r0+0x4];	lw r2, r0, 0x4
3	r3 ← r1 + r2;	add r3, r1, r2
4	M[r0+0x12] ← r3;	sw r3, r0, 0x12
5	r4 ← M[r0+0x8];	lw r4, r0, 0x8
6	r5 ← r1 + r4;	add r5, r1, r4
7	M[r0+0x16] ← r5;	sw r5, r0, 0x16



- Except for the instruction 3 and 6 other hazards can be resolved by data forwarding.

9

- We can use data forwarding technique to resolve most of the data hazard (as marked in green arrows in this slide). However, we still need to resolve dependency of EXE of instruction 3 on WB of instruction 2. Also we need to resolve dependency of EXE of instruction 6 on WB of instruction 5.

Data Hazard Solution - Reordering

Instruction		Pipeline Stages										
ID#	Statement	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
1	lw r1, r0, 0x0	IF	ID/RF	EXE	MEM	WB						
2	lw r2, r0, 0x4		IF	ID/RF	EXE	MEM	WB					
3	add r3, r1, r2			IF	ID/RF	EXE	MEM	WB				
4	sw r3, r0, 0x12				IF	ID/RF	EXE	MEM	WB			
5	lw r4, r0, 0x8					IF	ID/RF	EXE	MEM	WB		
6	add r5, r1, r4						IF	ID/RF	EXE	MEM	WB	
7	sw r5, r0, 0x16							IF	ID/RF	EXE	MEM	WB

Data Hazard & Resolution (Original)			
STAGE	DEPENDENCY	RESOLUTION	FWD-FROM
3-EXE	1-WB	FWD	1-MEM
3-EXE	2-WB	STALL	
4-MEM	3-WB	FWD	3-EXE
6-EXE	5-WB	STALL	
7-MEM	6-WB	FWD	6-EXE

10

- These tables shows formalization of instruction-stage dependency and resolution encoding. Colored registers and arrows correspond to each other to show dependency. For example red 'r1' shows that data origination of 'r1' is at instruction ID 1 and data consumption of 'r1' is at instruction ID 3 and 6. The corresponding arrow shows details of what stage of data consumption instruction is depending on what stage of data origination. We encode stage of an instruction by <instruction ID>-<stage short form>, where 'stage short forms' are IF, ID/RF, EXE, MEM, WB. To guarantee correct functionality, it is needed that data committing stage (1-WB in this case) of data origination instruction is completed before data retrieval stage (3-ID/RF in this case) of data consumption instruction. The arrow is drawn from completion of data committing stage (1-WB) to beginning of data consumption stage (3-EXE in this case).

- In the data hazard and resolution table we show exact data consumption stage (3-EXE for example) and data committing stage (1-WB) for example. Usually the 'DEPENDENCY' column should contain only X-WB, where X is instruction ID. The 'STAGE' column in this table is either 'X-EXE' or 'X-MEM' (memory write). Data is consumed only in these two stages. In the 'RESOLUTION' columns entries are either 'FWD' (for data forward) or 'STALL' (for stall). The 'FWD-FORM' column is relevant only when 'RESOLUTION' is 'FWD' . This column contains data origination stage which is either X-EXE or X-MEM (memory read).

Data Hazard Solution - Reordering

Data Hazard & Resolution (Original)			
STAGE	DEPENDENCY	RESOLUTION	FWD-FROM
3-EXE	1-WB	FWD	1-MEM
3-EXE	2-WB	STALL	
4-MEM	3-WB	FWD	3-EXE
6-EXE	5-WB	STALL	
7-MEM	6-WB	FWD	6-EXE

Instruction		Pipeline Stages (Resolution after Fwd Stall - '0' is stalled state)												
ID#	Statement	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
1	lw r1, r0, 0x0	IF	ID/RF	EXE	MEM	WB								
2	lw r2, r0, 0x4		IF	ID/RF	EXE	MEM	WB							
3	add r3, r1, r2			IF	ID/RF	0	EXE	MEM	WB					
4	sw r3, r0, 0x12				IF	0	ID/RF	EXE	MEM	WB				
5	lw r4, r0, 0x8					0	IF	ID/RF	EXE	MEM	WB			
6	add r5, r1, r4							IF	ID/RF	0	EXE	MEM	WB	
7	sw r5, r0, 0x16								IF	0	ID/RF	EXE	MEM	WB

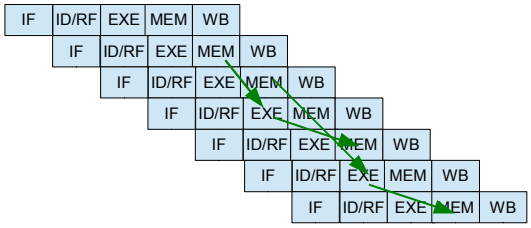
11

- It is usually one cycle stall that can resolve data hazard. Usually once stall is applied successive data hazards can be resolved using data forward technique. When a data hazard is detected and resolution is determined to be 'stall' the pipe line is stalled for one cycle at the beginning of data consumption stage. As a result successive instructions are also suffered from stall. In this example of stalling 3-EXE has introduced bubble state ('0') at 4-ID/RF and 5-IF. Pipeline is stalled not only for 3-EXE but also for 4-ID/RF and 5-IF.

- Data forwarding is shown from completion of data origination stage to start of data consumption stage.

Data Hazard Solution - Reordering

1	r1 ← M[r0+0x0];	lw r1, r0, 0x0
2	r2 ← M[r0+0x4];	lw r2, r0, 0x4
5	r4 ← M[r0+0x8];	lw r4, r0, 0x8
3	r3 ← r1 + r2;	add r3, r1, r2
4	M[r0+0x12] ← r3;	sw r3, r0, 0x12
6	r5 ← r1 + r4;	add r5, r1, r4
7	M[r0+0x16] ← r5;	sw r5, r0, 0x16



- By reordering instruction, data hazard can be solved (in this case) without stalling pipeline (data forwarding can resolve all the existing data hazard in this case).
  - bring instruction 5 to 3rd place.

12

- Compiler can re-order the instructions and bring the 5th instruction (another independent 'lw' instruction) to 3rd place. There will be no result change for this example if this is done. With this reordering of instruction, we are pushing instruction 3 one more step away from instruction 2 in the pipeline. As a result data hazard between 2 and 3 can be resolved by data forwarding. Similarly, with this re-ordering , we are also separating instruction 5 and 6 by 2 steps in the pipeline. Hence, the data hazard between them can also be resolved by data forwarding.

- Compilers are architecture dependent and has knowledge of such data dependencies and hazard. It tries its best to avoid stalling (which is a hardware feature to identify hazard and stall pipeline. This feature is implemented in controller of the pipe-lined processor) by re-ordering the instructions.

Data Hazard Solution - Reordering

Instruction		Pipeline Stages (Reordered)										
ID#	Statement	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
1	lw r1, r0, 0x0	IF	ID/RF	EXE	MEM	WB						
2	lw r2, r0, 0x4		IF	ID/RF	EXE	MEM	WB					
5	lw r4, r0, 0x8			IF	ID/RF	EXE	MEM	WB				
3	add r3, r1, r2				IF	ID/RF	EXE	MEM	WB			
4	sw r3, r0, 0x12					IF	ID/RF	EXE	MEM	WB		
6	add r5, r1, r4						IF	ID/RF	EXE	MEM	WB	
7	sw r5, r0, 0x16							IF	ID/RF	EXE	MEM	WB

Data Hazard & Resolution (Reordered)			
STAGE	DEPENDENCY	RESOLUTION	FWD-FROM
3-EXE	1-WB	FWD	1-MEM
3-EXE	2-WB	FWD	2-MEM
4-MEM	3-WB	FWD	3-EXE
6-EXE	5-WB	FWD	5-MEM
7-MEM	6-WB	FWD	6-EXE

13

- The yellow highlighted instruction (ID 5) has been moved up into order of instruction. This instruction being picked up for being independent of prior instructions. With this re-ordering, we can see that the instruction 3 is spaced further by one more clock cycle from instruction 2 (which was causing stall without reordering) and instruction 6 is spaced further by 2 more clock cycle from instruction 5 (which was causing another stall without reordering) . In this case, reordering of just one instruction eliminated two stalls.

Data Hazard Solution - Reordering

Data Hazard & Resolution (Reordered)			
STAGE	DEPENDENCY	RESOLUTION	FWD-FROM
3-EXE	1-WB	FWD	1-MEM
3-EXE	2-WB	FWD	2-MEM
4-MEM	3-WB	FWD	3-EXE
6-EXE	5-WB	FWD	5-MEM
7-MEM	6-WB	FWD	6-EXE

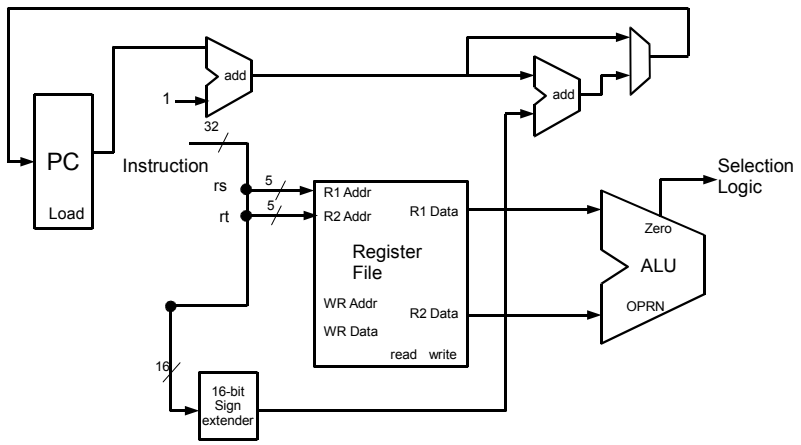
Instruction		Pipeline Stages (Reordered - Resolution)										
ID#	Statement	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
1	lw r1, r0, 0x0	IF	ID/RF	EXE	MEM	WB						
2	lw r2, r0, 0x4		IF	ID/RF	EXE	MEM	WB					
5	lw r4, r0, 0x8			IF	ID/RF	EXE	MEM	WB				
3	add r3, r1, r2				IF	ID/RF	EXE	MEM	WB			
4	sw r3, r0, 0x12					IF	ID/RF	EXE	MEM	WB		
6	add r5, r1, r4						IF	ID/RF	EXE	MEM	WB	
7	sw r5, r0, 0x16							IF	ID/RF	EXE	MEM	WB

- This table shows data forwarding after the instruction reordering.

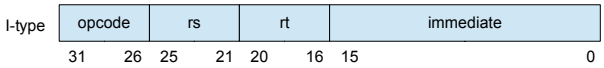
Pipelining Control Hazard ...

15

Review of Branch Instruction Data Path



Operation: PC = PC + 1 + SignExtImm;
if R[rs] == R[rt] or R[rs] != R[rt]



16

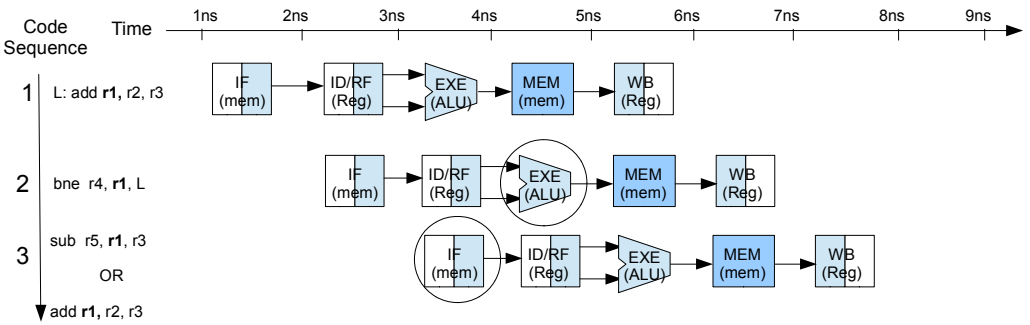
- From Lecture 11 / slide 24.

- For I-type branch operations branch address is computed by adding PC+1 with the sign extended immediate filed (this enables jump forward and backward from one decision point). The selection of next address between (PC+1) or (PC+1+SignExtImm) is done with a mux with selection control signal generated for Branch-On-Equal and Branch-On-NoEqual. (PC+1+SignExtImm) is done with an extra adder.

- The content of 'rs' and 'rt' register are subtracted and zero flag is tested. If the zero flag is on, the content of 'rs' and 'rt' are equal, otherwise not. ALU unit usually provides the zero flag too. If it does not, it is easy to implement by doing a NOR between all the bits of the result. Only if all the bits are zero, then the zero flag will be turned on.

- If ZeroFlag is true and operation is 'beq' then (PC+1+SignExtImm) is selected. On the other hand, if ZeroFlag is false and operation is 'bne' then (PC+1+SignExtImm) is selected. (PC+1) is selected in all other cases.

Control Hazard

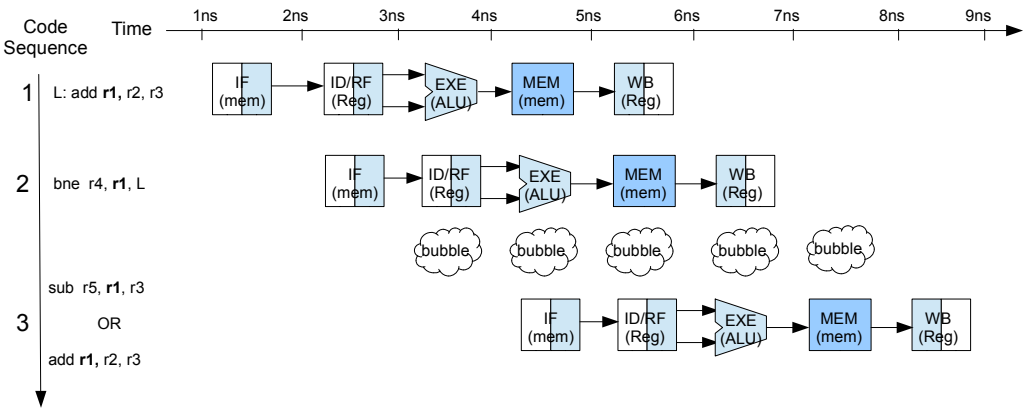


- If the branch is taken or not can not be concluded till EXE stage is done for instruction 2.

17

- In this example, the 3rd instruction to be executed depends on whether r[1] and r[4] are equal or not. If they are equal the 3rd instruction executed will be 'sub r[1], r[3], r[5]' and if they are not, processor should execute instruction 1. Since whether r[1] and r[4] are equal or not can only be known after EXE stage of instruction 1, it is not possible to determine what will be the instruction to be fetched for 3rd instruction fetch. This results in control hazard, where next instruction location is indeterminstic.

Control Hazard Solution - Stalling



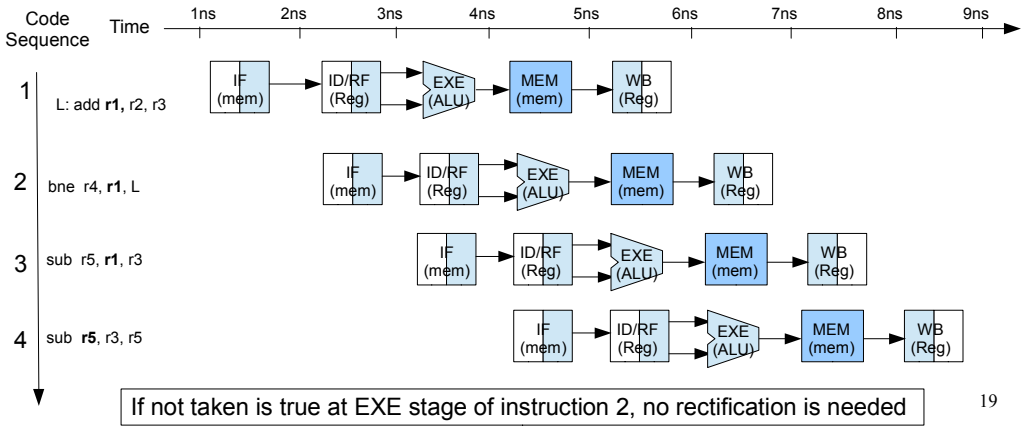
- Stall the pipeline for one cycle.

18

- One obvious way to resolve control hazard is to stall the pipeline till comparison result is available. However, program usually has many such branching instruction. Hence by stalling the pipeline we can significantly reduce the performance of the process.

Control Hazard Solution - Prediction

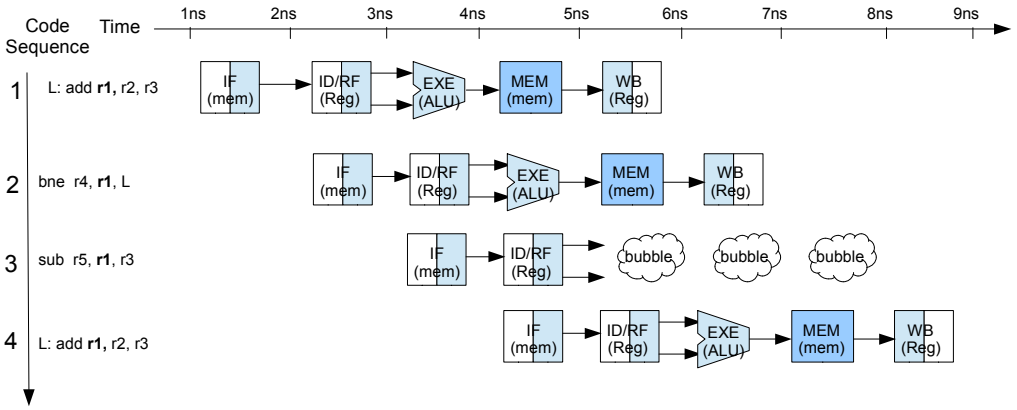
- Predict branch is taken or not taken.
  - e.g. always predict not taken.
  - As soon as decision is made, if wrong prediction, abort the latest instruction from the pipeline.



19

- One way to avoid stalling is to predict branch taken or not taken. In this case, if branch not taken is always predicted, then the 3rd instruction fetched will be the 'sub r[1] r[3] r[5]'. If the test in instruction 2 is false at EXE stage of it, there is no problem, because in that case, the next instruction to be fetched is the instruction 'sub r[1] r[3] r[5]' (which is already being fetched and is in pipeline). It can continue to fetch even the next 4th instruction (sub r[5] r[3] r[5]).

Control Hazard Solution - Prediction

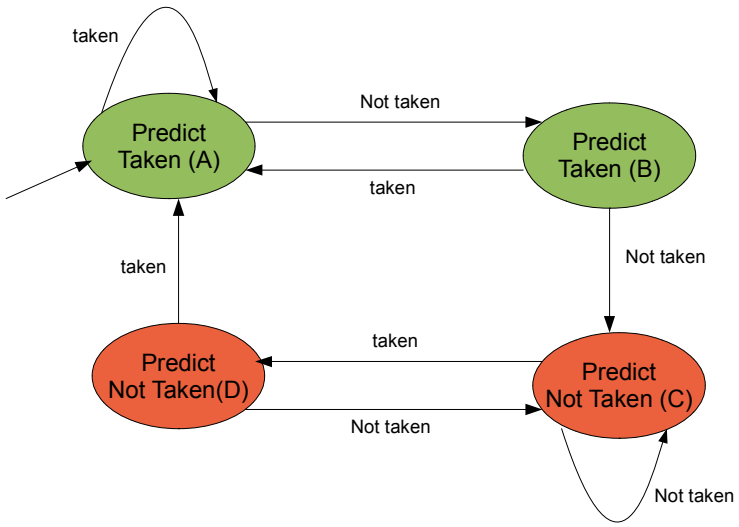


If not taken is false at EXE stage of instruction 2, instruction 3 is aborted and 4th instruction executed is instruction 1.

20

- However, if the test at instruction 2 turned out to be true, then we need to abort 3rd instruction (sub r[1] r[3] r[5]) and fetch instruction 1 as the 4th instruction. We abort the 3rd instruction by inserting no operation for rest of the stages [mechanism to do so is beyond scope of this course]. This abortion of instruction should not effect on the final result of the program, since the 3rd instruction result is not committed to data memory or register file at the abortion time.

Dynamic Probabilistic Prediction



21

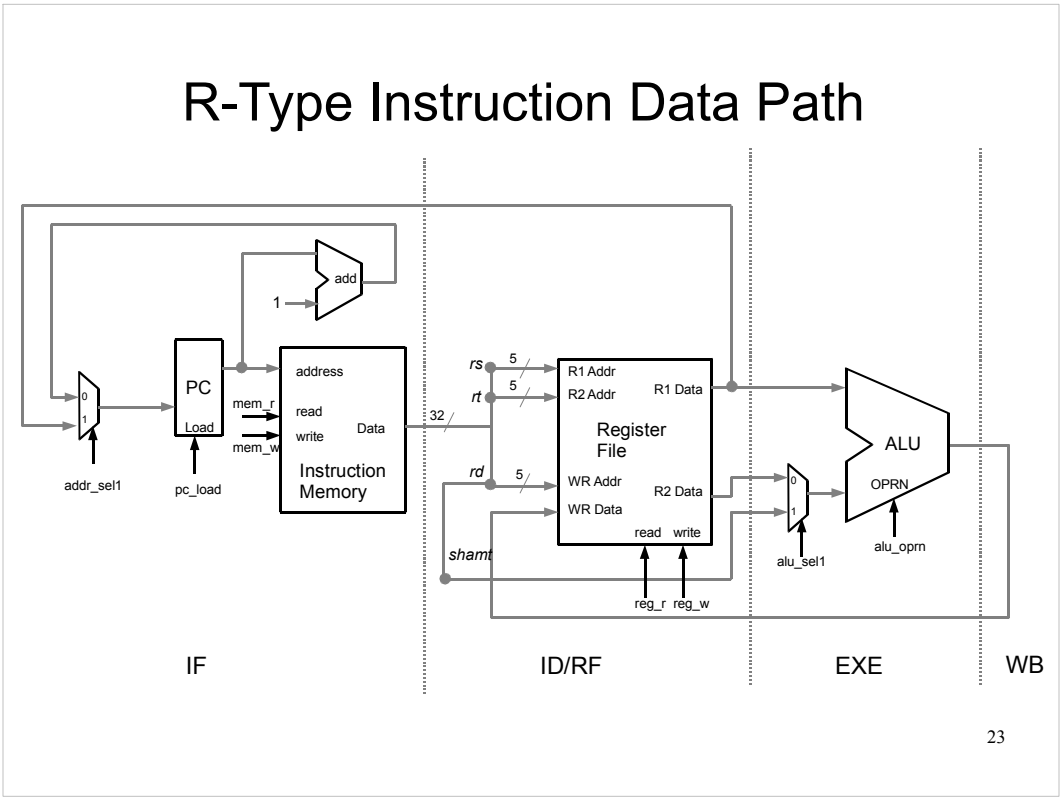
- So far, we have discussed a static prediction for 'branch taken' or 'not taken'. Usually modern processor implements dynamic probabilistic prediction, where branch prediction depends on the number of time if the branch is taken or not taken during the execution.

- In this strategy, branch prediction controlled enters into a known state of 'predict taken' (A) or 'predict not taken' (C) (depending designers choice). Let's say the initial state of prediction is 'branch taken' (A).
  - State A denotes that 'branch taken' is done so far and next prediction is 'branch taken'.
  - State B denoted that one 'branch not taken' is done and next prediction is 'branch taken'.
  - State C denotes that 'branch not taken' is done so far and next prediction is 'branch not taken'.
  - State D denotes that one 'branch taken' is done and next prediction is 'branch not taken'.

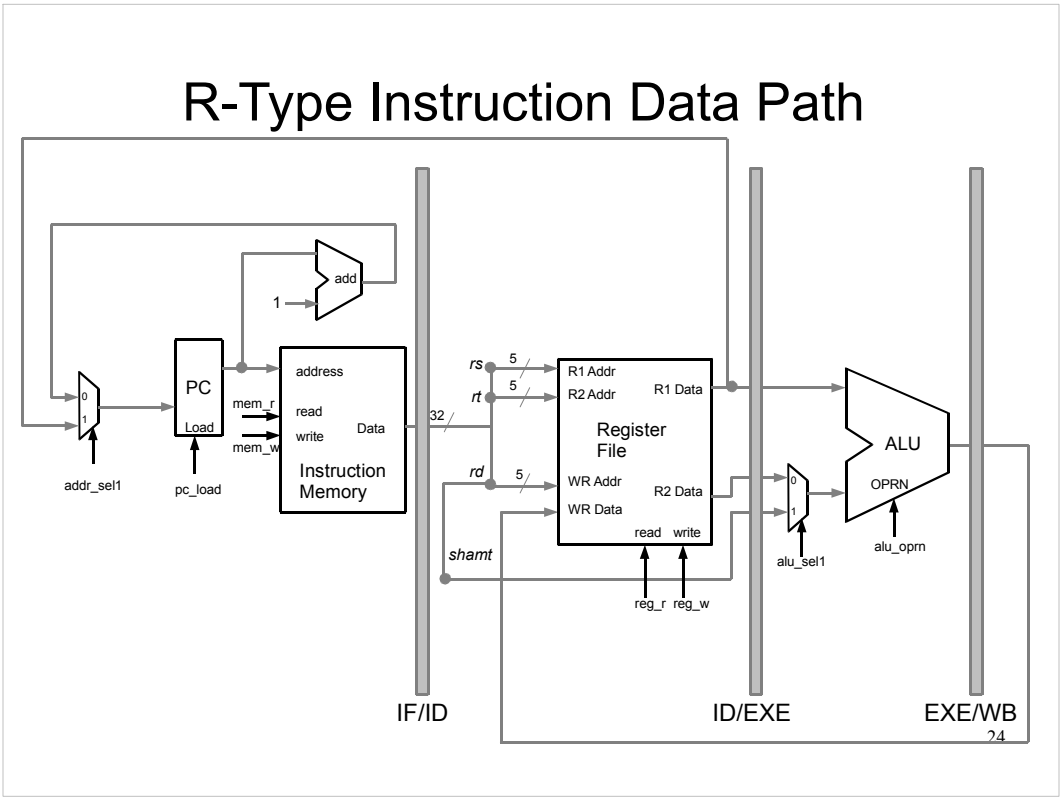
- State transitions are done in the following manner.
  - A → A : if a branch is taken.
  - A → B : if a branch is not taken.
  - B → A : if a branch is taken.
  - B → C : if a branch is not taken.
  - C → C : if a branch is not taken.
  - C → D : if a branch is taken.
  - D → C : if a branch is not taken.
  - D → A : If a branch is taken.

- This type of dynamic prediction will reduce probability of aborting an instruction in pipeline. For example, in a for-loop, it is more likely that the branch will be taken for most of the time. Hence if it is a static prediction of 'branch not taken' almost all the time an instruction in the pipeline needs to be aborted. On the otherhan, if the dynamic prediction is done, the prediction is somewhat depending on the input data set of the program which controls the behavior of the loops and decisions and thus can give more efficient branching strategy which minimizes number of instruction abortion.

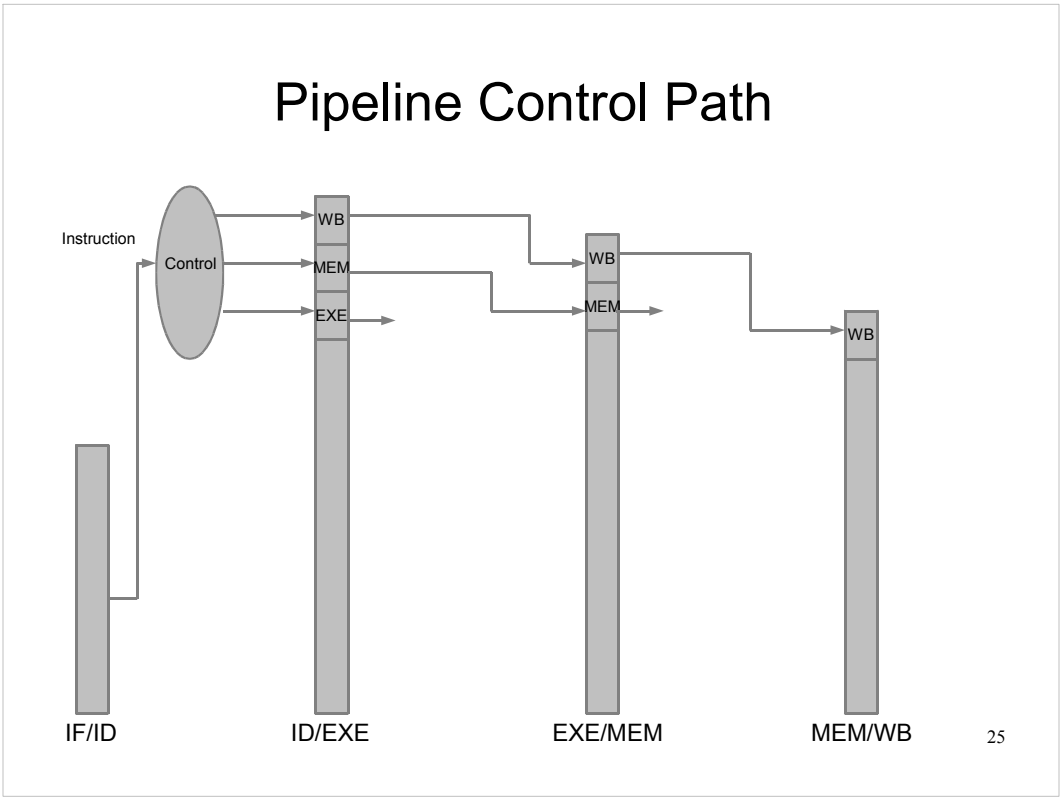
Pipeline Implementation ...



- The data path (R-type in this example) can be divided into pipeline stages. The PC, Instruction memory, next address calculator and the address selection can be grouped as data path components needed for instruction fetch stage. Similarly register file is involved in register fetch / instruction decode stage. The data selection mux and ALU is involved in EXE stage and the register file is also involved in WB stage. For R-type instructions, there is not data memory involved. For I-type and J-type instructions, data memory is involved in the MEM stage of pipe line.



- We put registers at the boundary of different groups of components for different pipe line stages. For a 5-stage pipeline architecture there will be 4 such registers to transfer data between IF/ID, ID/EXE, EXE/MEM and MEM/WB stage. At each clock cycles data is transferred from one stage to another. To implement stall, data loading is disabled into these boundary registers (remember, registers has a control signal 'load' which enable loading of new data into registers at clock edge). For example, if we disable load signal at IF/ID boundary, the current instruction will not flow through the pipeline, but the previous instruction will proceed for completion through rest of the pipeline.



- The control signals for EXE, MEM and WB stage can be generated after instruction fetch is done. Once they are generated at the ID/RF stage, they also flows through boundary registers ID/EXE, EXE/MEM and MEM/WB and consumed at appropriate stage. For example, the EXE stage control signal will be consumed at the EXE stage of pipeline. After EXE stage, WB and MEM control signal will continue to be transferred to successive stages.

CS147 - Lecture 15

Kaushik Patra
(kaushik.patra@sjsu.edu)

26

- Pipeline Data Hazard
- Pipeline Control Hazard
- Pipeline Implementation

Reference Books:

1) Chapter 6 of 'Computer Organization & Design' by Patterson & Hennessy