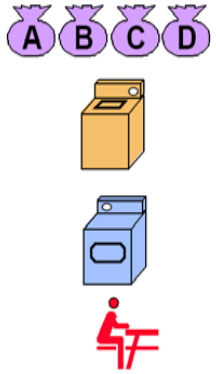
# Pipelining

CIT 595
Spring 2007

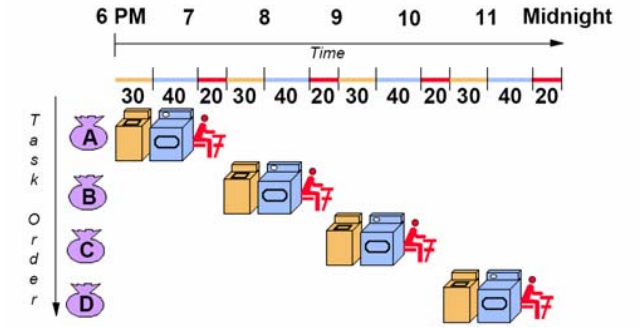
## Laundry Example

- Ann, Brian, Cathy and Dave each have one load of clothes to wash, dry and fold
- Washer takes 30 mins
- Dryer takes 40 mins
- "Folder" takes 20 mins





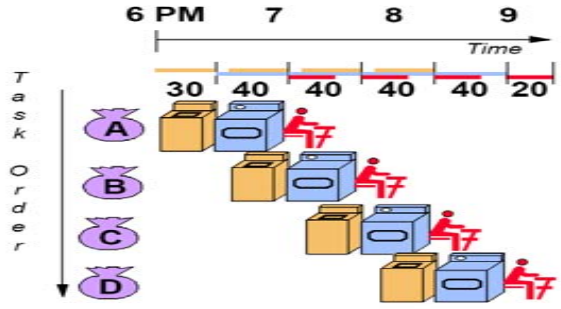

## Sequential Laundry




- Entire workload takes 6 hours to complete



## Pipelined Laundry




- Pipelined Laundry takes only 3.5 hours
- Speedup = 6/3.5 = 1.7
- Pipelining did not reduce completion time for one task but it helps the throughput of the entire workload in turn decreasing the completion time

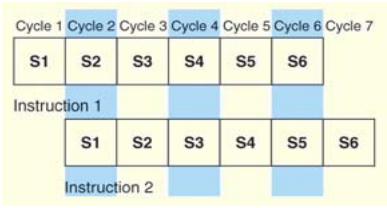
## Instruction Level Pipelining

- Pipelining is also applied to Instruction Processing
- In instruction processing, each instruction goes through F->D->EA->OP->EX->S cycle
- The instruction cycle is divided into stages
  - One stage could contain more than one phase of the instruction cycle or one phase can be divided into two stages
- If an instruction is in a particular stage of the cycle, the rest of the stages are **idle**
- We exploit this idleness to allow instructions to be executed in parallel
- From the Laundry Example, we know that throughput increase also allows reduction in completion time, hence overall program execution time can be lowered
- Such parallel execution is called *instruction-level pipelining*



## Instruction Level Pipelining: Big Picture

- Each stage of the Instruction Processing Cycle takes 1 clock cycle
  - 1 clock cycle = *x* time units per stage
- For each stage, one phase of instruction is carried out, and the stages are overlapped




S1. Fetch instruction
S2. Decode opcode
S3. Evaluate Address
S4. Fetch operands
S5. Execute
S6. Store result



## Theoretical Speedup due to Pipelining

The theoretical speedup offered by a pipeline can be determined as follows:

- Let *k be total number of stages and* $t_p$ be the time per stage
- Each instruction represents a task, *T*, in the pipeline and *n* be the total number of tasks
- The first task (instruction) requires $k \times t_p$ time to complete in a *k*-stage pipeline.
- The remaining (*n* - 1) tasks emerge from the pipeline one per cycle
- So the total time to complete the remaining tasks is $(n - 1)t_p$
- Thus, to complete *n* tasks using a *k*-stage pipeline requires:

  $(k \times t_p) + (n - 1)t_p = (k + n - 1)t_p$



## Theoretical Speedup due to Pipelining

If we take the time required to complete *n* tasks without a pipeline and divide it by the time it takes to complete *n* tasks using a pipeline, we find:

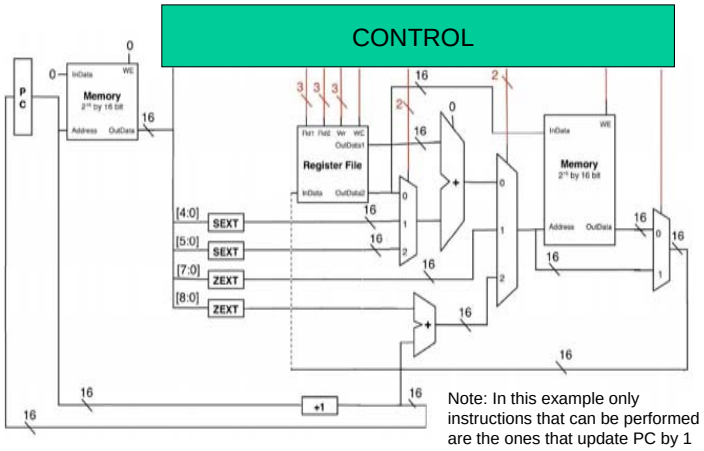
$$\text{Speedup } S = \frac{n t_n}{(k + n - 1) t_p}$$

$t_n = k \times t_p$

If we take the limit as *n* approaches infinity, (*k* + *n* - 1) approaches *n,* which results in a theoretical speedup of:

$$\text{Speedup } S = \frac{k t_p}{t_p} = k$$

## LC3 Hardwired Control (modified)




Note: In this example only instructions that can be performed are the ones that update PC by 1



## How Pipelining actually Implemented??

- Since we are overlapping stages (with the exception of Fetch stage), all the control information plus data (i.e. information along the data path) must be remembered per instruction and must be carried through each stage

- This is achieved by placing a n-bit register that can hold the control/data information in between each stage



## LC3 Pipelined Implementation

With reference to diagram of Hardwired Control LC3 Implementation

Note:

- Since Evaluate Address and Execute both use ALU, we can make this one stage

- The Operand Fetch is separated into Register Fetch and Memory Access (one phase is split into two stages)

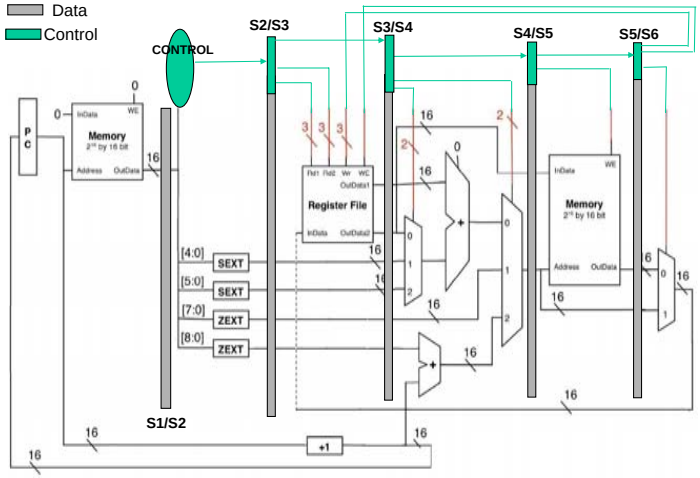
- Store consists of only register writes (not memory writes)

- Memory Write is part of Memory Access

- Thus we have a total of 6 stages



## LC3 Pipelined Implementation



## Impact on Clock Cycle due to Pipelining

• Again for pipelining, the *clock* is sequencing the stages (instructions move in lock step fashion)

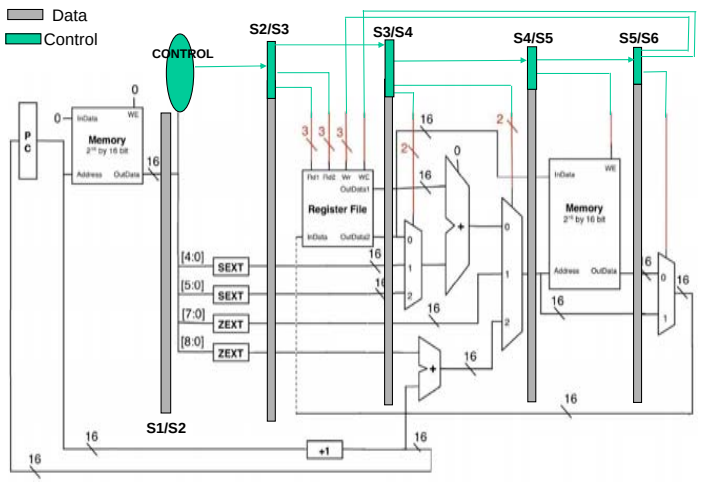
• For pipelining to work correctly, we want to make sure that all work done in one stage gets done on time before it moves to next stage

• Hence, the clock cycle time should be as long as time it takes through the longest pipe stage (this also includes the time for capturing data into registers in between stages)

$$\text{Clock cycle time} = \max(t_1, t_2, t_3, t_4, t_5, t_6)$$



## LC3 Pipelined Implementation






## Impact on Clock cycle time due to Pipelining

• Recall

$$\text{CPU Time} = \frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}}$$

$\frac{\text{time}}{\text{cycle}}$: **Clock Cycle time**; $\frac{\text{cycles}}{\text{instruction}}$: CPI; $\frac{\text{instructions}}{\text{program}}$: Instruction Count

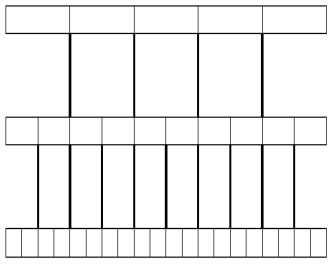
• If we lower the time per cycle, this will lower the program execution time and hence improve performance

• This implies that we if we shorten the time per pipeline stages, we will lower clock cycle time

➢ This can be achieved by adding more pipe stages of shorter duration



## Impact on Clock cycle time due to Pipelining



5 stage

10 stage – clock cycle time reduced by one half

20 stage – clock cycle time reduced by one fourth

## Cycles Per Instruction (CPI) with Pipelining

• In pipelining, one instruction is in each stage

• Since one instruction will be fetched (or finish) each cycle, the average CPI will equal 1 (obviously we are ignoring the very first instruction – cold start)


• However, CPI = 1 is barely achieved



## Why CPI is not always 1?

We assume that the pipeline can be kept filled at all times

• However, this is not always the case

The situations that cause pipeline not to filled at all times arises due to what is known as *Pipeline Hazards*



## Pipeline Hazards

There are three kinds of pipeline hazards:

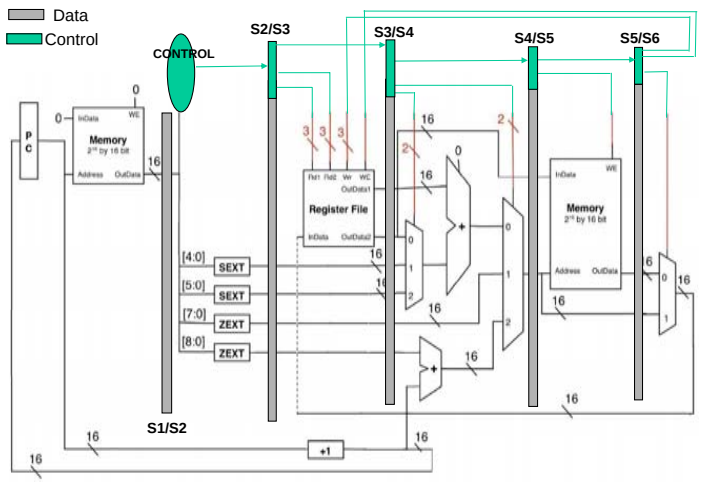
1. Structural Hazard
2. Data Hazard
3. Control Hazard



## Structural Hazard

- Occurs when hardware cannot support a combination of instructions that we want to execute in parallel
  - In Laundry example: the machine has combined washer or dryer
- In instruction pipelining, it usually occurs when one hardware is shared amongst two stages that work in parallel
  - Example: Memory or Register File
- Usually overcome by duplicating hardware
  - Memory is separated into instruction and data memory
  - Or memory/register is multi-ported i.e. memory that provides more than one access path to its contents

## LC3 Pipelined Implementation






## Data Hazard

- Occurs when an instruction depends on the results of a instruction still in the pipeline
- Example 1:
  i1: ADD R1, R2, R3
  i2: AND R5, R1, R4
- Example 2:
  i1: ADD R1, R2, R3
  i2: ST R1, A
- Example 3:
  i1: LD R1, A
  i2: ADD R2, R1, R2



## Data Hazard: Example 1

i1: ADD R1, R2, R3
i2: AND R5, R1, R4

S1: Instruction Fetch
S2: Decode
S3: Register Fetch
S4: Execute/EA
S5: Memory Access (LD/ST)
S6: Write Back (register write)

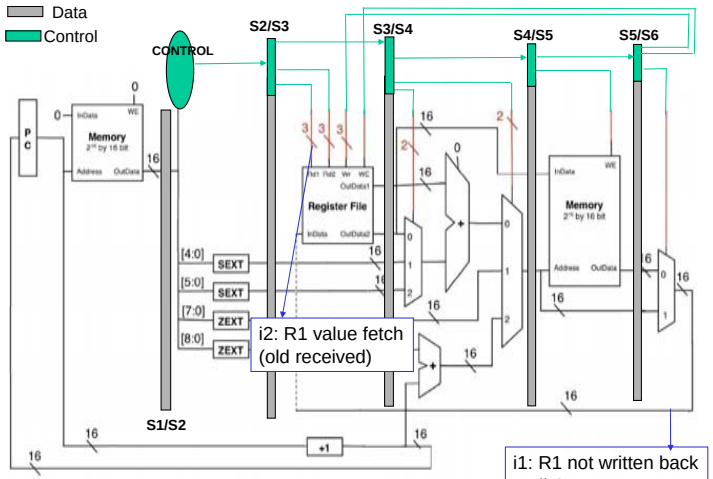
| cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| i1 | S1 | S2 | S3 | S4 | S5 | S6 | | |
| i2 | | S1 | S2 | S3 | S4 | S5 | S6 | |

i2 fetching R1, gets stale value

i1 completed i.e. value is written to R1



## Data Hazard Example 1



## Solution to Example 1

- Naive approach, introduce delay in the pipeline till instruction i1 finishes
- Also stop any new instructions from being fetched
- Also known as Pipeline Stall or Bubble

| cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| i1 | S1 | S2 | S3 | S4 | S5 | S6 | | | |
| i2 | | S1 | S2 | .. | .. | … | S3 | S4 | S5 | S6 |

S1: Instruction Fetch
S2: Decode
S3: Register Fetch
S4: Execute/Evaluate Addr
S5: Memory Access (LD/ST)
S6: Write Back (register write)



## Inserting delay in the Pipeline?

- As instructions are fetched and decoded, control logic/special hardware determines whether a hazard could/will occur

- If this is true, then the control logic
  - Generates control such that next instruction will not be fetched and
  - Suspends the instruction that will cause the hazard
    - Suspension is nothing but disabling all the stages till a few cycles such that nothing occurs in them (known as inserting NOP i.e. No-operation)

- This provides the instruction before the hazard instruction sufficient time to complete and hence prevent the hazard



## Solution to Example 1 (contd..)

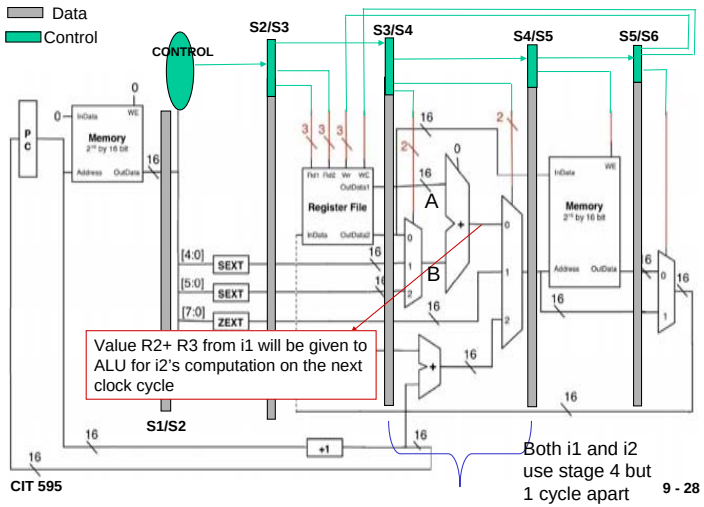
- Better Solution: Data Forwarding
  - Realize that data value from i1 (to be put in R1) is actually available at end of cycle 4
  - Don't need to wait till S7 to fetch the register file, instead forward a copy of the data from S4 of i1 to i2's stage S4

| cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| i1 | S1 | S2 | S3 | S4 | S5 | S6 | |
| i2 | | S1 | S2 | S3 | S4 | S5 | S6 |

S1: Instruction Fetch
S2: Decode
S3: Register Fetch
S4: Execute/EA
S5: Memory Access (LD/ST)
S6: Write Back (register write)



## Data Hazard Example 1: Forwarding



## Handling Data Forwarding

Requires additional logic to data path

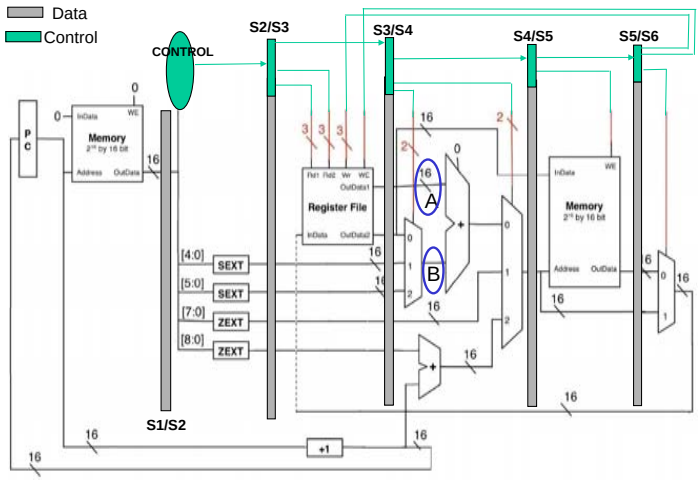
- Additional MUX needs to be place to select between output of the register file and forwarded input for input *A* to ALU

- The control will have additional task to figure whether there is hazard condition and accordingly set the MUX control

- Input *B to ALU*, the existing MUX will needed to be expanded
  - i.e. i1: ADD R1 R2, R3
    i2: ADD R5, R4, R1



## Data Path changes due to Forwarding

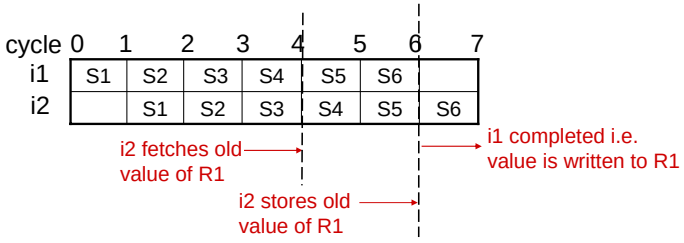





## Data Hazard: Example 2

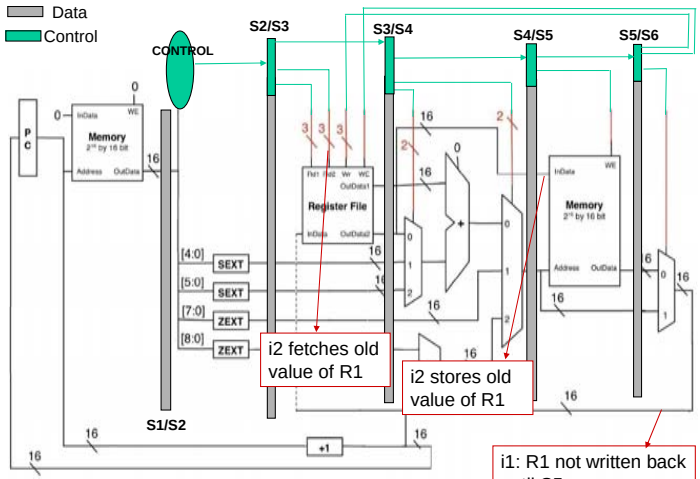
i1: ADD R1, R2, R3
i2: ST R1, A

S1: Instruction Fetch
S2: Decode
S3: Register Fetch
S4: Execute/Evaluate Addr
S5: Memory Access (LD/ST)
S6: Write Back (register write)






## Data Hazard Example 2



## Solution to Example 2

• Stall the Pipeline

| cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| i1 | S1 | S2 | S3 | S4 | S5 | S6 | | | | |
| i2 | | S1 | S2 | .. | .. | .. | S3 | S4 | S5 | S6 |

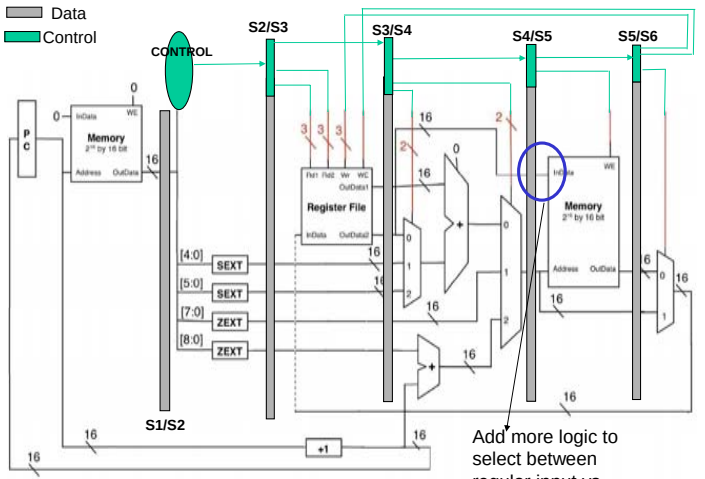
• Forwarding

- Realize that data value (to be put in R1) is actually available at end of cycle 4 and is also propagated through to next stage
- Don't need to wait till cycle 6, forward a copy of the data to i2's stage S5

| cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| i1 | S1 | S2 | S3 | S4 | S5 | S6 | | |
| i2 | | S1 | S2 | S3 | S4 | S5 | S6 | |



## Example 2 Data Forwarding






## Data Hazard: Example 3

i1: LD R1, A
i2: ADD R2, R1, R2

S1: Instruction Fetch
S2: Decode
S3: Register Fetch
S4: Execute/Evaluate Addr
S5: Memory Access (LD/ST)
S6: Write Back (register write)

| cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| i1 | S1 | S2 | S3 | S4 | S5 | S6 | | |
| i2 | | S1 | S2 | S3 | S4 | S5 | S6 | |

i2 fetching R1, gets stale value

i1 completed i.e. value is written to R1



## Solution to Example 3

• Complete Data Forwarding not possible in this case

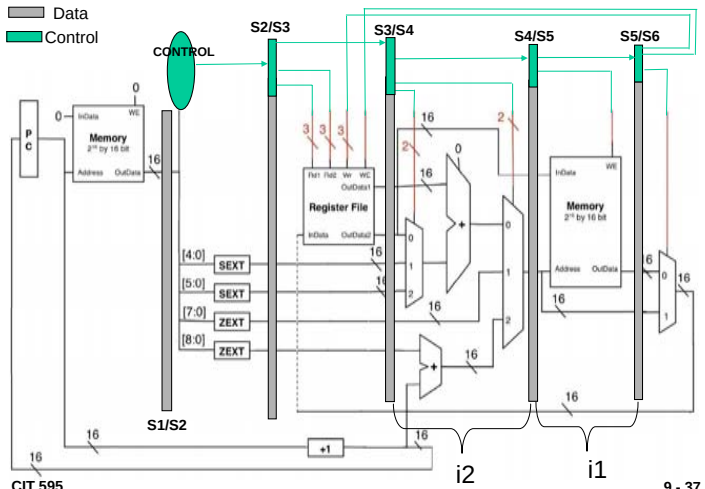
| cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| i1 | S1 | S2 | S3 | S4 | S5 | S6 | | |
| i2 | | S1 | S2 | S3 | S4 | S5 | S6 | |

Value from memory (to be put in R1) is received from memory at end of cycle 5 for i1, but i2 needs value of R1 at beginning of cycle 4

• Stall for one cycle and then Forward

| cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| i1 | S1 | S2 | S3 | S4 | S5 | S6 | | | |
| i2 | | S1 | S2 | S3 | .. | S4 | S5 | S6 | |

## Example 3

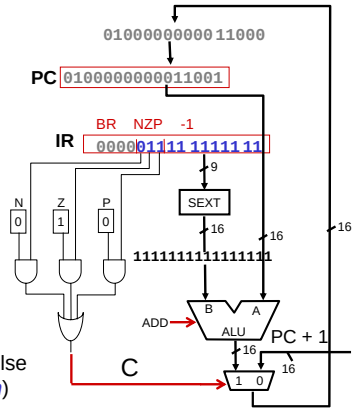



load-use hazard



## Recap: Branch Instruction

NZP registers are set by NZP logic




If C = 1, then Branch Target Address (i.e. *Branch Taken*) else PC + 1 (i.e. *Branch Not Taken*)



## Control Hazards

- Occurs when we need to make a decision based on the result of instruction while others are executing

- Branch Instructions are instructions that make decision
  - ➢ Alter the flow of execution in a program and give rise to control hazard



## Control Hazard

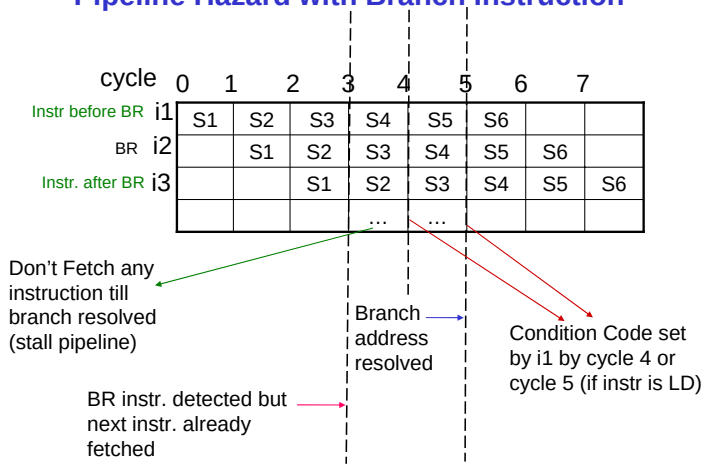
- The problem with branch instructions is that:

1. We find out that instruction is Branch Instr only after *Decode* Stage (S2) and by then we have already fetched the next sequential instr.

2. Branch address is resolved only in the *Evaluate Address* phase (S4)
   - ➢So we have to stall the pipeline till we know which address to fetch from i.e. PC + 1 or Branch Target address

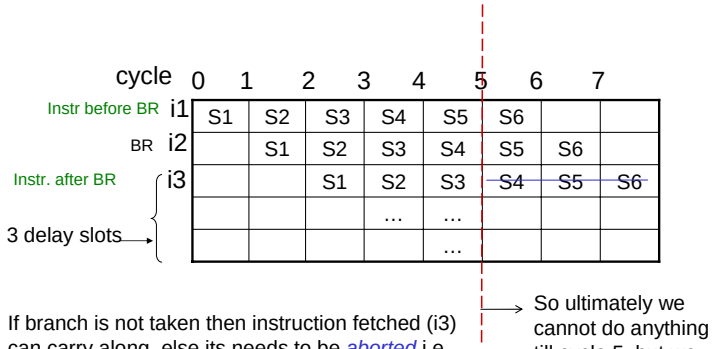
3. The instr. before the branch instr. will set *Condition Code* (NZP) one cycle before or the same cycle the branch address is resolved

## Pipeline Hazard with Branch Instruction






## Pipeline Hazard with Branch Instruction (contd..)




If branch is not taken then instruction fetched (i3) can carry along, else its needs to be *aborted* i.e. no memory elements should be updated (requires additional circuitry for doing this)



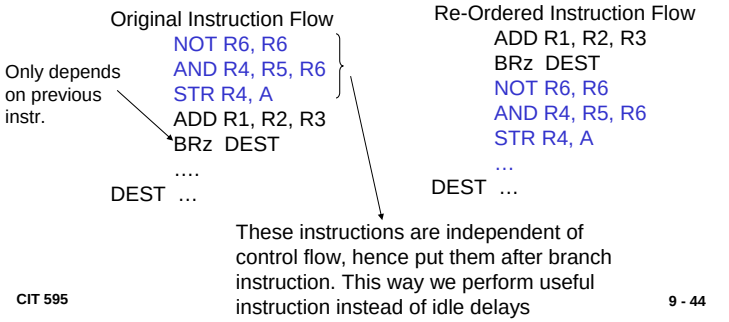
## Branch Instruction Impact CPI

- Hazards increase the # cycles/instr. in pipelined implementation
- Structural and Data Hazard effects can be minimized
- However, branch hazards cannot be minimized because we have to wait for following information
  - Branch address (tells us where to branch)
  - Condition Code from the previous instruction (tells us whether to branch or not)
- Most ISAs have some sort of pipelined implementation
  - Many techniques have been studied to *reduce branch delays*



## Control Hazard Solution 1: Reduce the BR Delays

- ***Compiler*** resolves branching by rearranging the machine code to fill *delay slots* after BR instruction
  - By inserting ***useful instruction*** that can be done without changing the output of the program



## Control Hazard Solution 1: Reduce the BR Delays

- But if this is not possible then compiler will insert *NOP* instruction to keep the pipeline full
  - ISAs provide NOP instructions to insert delay
  - NOP does nothing, just there to kill some cycles
  - NOP instructions have all zeros in their bit fields
  - E.g. NOP opcode in LC3 would 0000 be (bits [15:12])

```
        ADD R1, R2, R3
        BRz  DEST
        NOP
        NOP
        NOP
        …
DEST    …
```



## Control Hazard Solution 2: Prediction

- Compiler can only help if they are enough instructions to re-order
- Many ISAs also often *predict* the outcome of the branch
  1. In these ISAs address calculation of branch is coupled in the same phase as the branch is discovered (i.e. moved from S4 to S2)
  2. There is *prediction unit* that records history of branch pattern
  3. Once the branch instruction is discovered, the prediction unit guides processor which instructions to fetch next

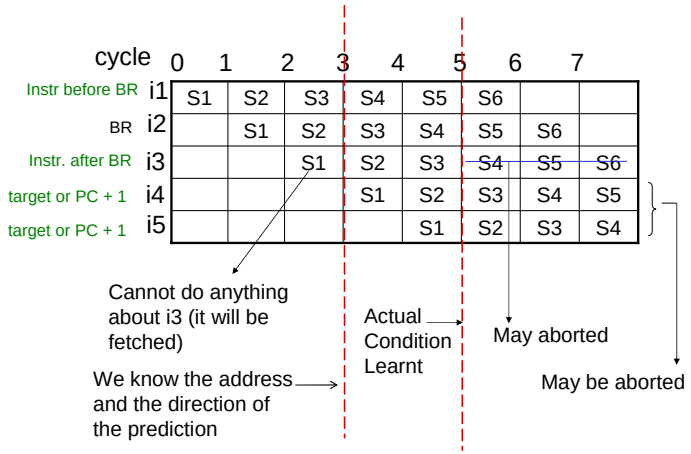


## Using Prediction Unit

- If prediction is Taken: the instruction that are fetched for the delay slot are from the target address path (also calculated in the same stage as BR discovered)
- If prediction is Not Taken: the instructions that are fetched for the delay slot are from PC + 1 path
- If actual result = prediction, don't do anything i.e. continue the processing
  Else need to abort the fetched instruction and restart
- Also update the prediction unit with the actual result i.e. out come the current branch instruction (for future branches)



## Pipelining with Branch Prediction



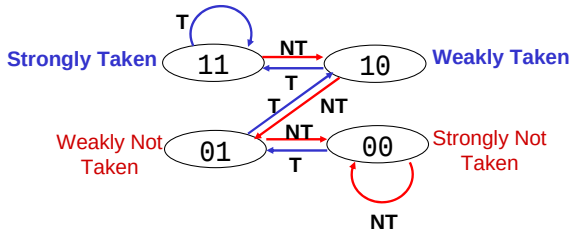
## Types of Prediction

- Static (wired/fixed)
  - Always guess “taken” or “not taken”
  - Effective only with loop control structure
- Dynamic
  - Another hardware in the datapath keeps tracks of the branch history as the instructions are executing
  - E.g. 2-bit branch predictor using saturating counters



## Example: Two-bit Branch Predictor

- Keep 2-bit history value for each “recent” BR instruction
- Use 2-bit saturating counter
  - If branch is actually Taken (T), increment the history value
  - If Not Taken (NT), decrement the history value
  - 00 (Strongly NT),01 (Weakly NT), 10 (Weakly T), 11 (Strongly T)






Typically > 90 % correct predictions



## Prediction helps reduce impact on Avg. CPI

- Each branch instruction takes 1 cycle to complete + additional cycles (due to branch delays caused)
- Lets say 20% instructions are branches
- Assume that branch predictor is 90% accurate
- **Pipeline without Prediction**

  $CPI_{branch}$ = Fraction Branches * (1 + Additional Cycles)
  = (0.2) * (1 + 3) = 0.8

- **Pipeline with Prediction**

  $CPI_{branch}$ = Fraction Branches * ( 1 + 1 + (Misprediction Rate * Additional Cycles))
  = (0.2) *(2 + (1 - .90) * 2)
  = 0.44

  Assume instr. right after BR always cause 1 delay (conservative)



## Compiler + Prediction helps reduce impact on Avg. CPI

- Lets say 20% instructions are branches
- Assume the compiler only finds one useful instruction on average
  - Hence additional cycles needed is 2 due to NOPs
- Pipeline with Compiler

  $CPI_{branch}$ = Fraction Branches * (1 + Additional Cycles)
  = (0.2) * (1 + 2) = 0.6

- Now, if we kick in the predictor (90% accurate) for 2 NOPs

  $CPI_{branch}$ = Fraction Branches * (1 + [Misprediction Rate * Additional Cycles])
  = (0.2) *[1 + [ (1 - .90) * 2])
  = 0.24

## JUMP Instruction is also Control Hazard

• Jumps don't have conditions i.e. jumps are unconditional branches (we will definitely go to the target address)

• Have to wait till we evaluate the address i.e. read address from register file (e.g. JMP R3)




## Delays due to JUMP instruction

• Additional delay of 2 cycles will be incurred
  - one instruction that will eventually be aborted and
  - one stall for not fetching next instruction after discovering Jump instruction

• Branch Prediction cannot help in this case as we are not waiting on a condition

• Any penalty if to be reduced will fall on compiler i.e. find useful instructions to do in the 2 delay slots



## Deeper Pipelines and Misprediction Penalty

• Dividing the pipeline into even smaller stages increases frequency (i.e. lowers time/cycle)

• But deeper the pipeline, will cause branch resolution to later stages, in turn increasing the CPI due misprediction penalty
  - If we filled pipeline with instructions from the wrong path then we wasted cycles for those instructions

• Hence the performance does not scale well with deeper pipelines



## Exceptions

Exceptions are used for signaling a certain condition

You already know
1. I/O request: device requests attention from CPU
2. System call or Supervisor call from software (TRAP in LC3, I/O functions in C)
3. Arithmetic: Integer or FP, overflow, underflow, division by zero
4. Invalid opcode: CPU was given an wrongly formatted instruction
5. Memory protection: read/write/execute forbidden on requested address

Yet to learn (or may be heard)
1. Page fault: requested virtual address was not present in main memory
2. Misaligned address: bus error
3. Hardware malfunction: component failure

## Handling I/O requests and System Calls

• I/O request either via system calls (TRAP) or interrupt signals

• Once we encounter such a request
  - Stop fetching further instructions
  - Complete all instructions in the pipeline ***before*** interrupt/sys call occurred
    - In the case of sys call abort instructions after sys call
  - Save the state of the processor
  - Jump to the servicing routine to handle interrupt/sys call

• In the case of interrupts we might have multiple requests, but the one with higher priority will be serviced
  - Separate hardware determines priority



## Handling All other Exceptions

• Let the instruction(s) ***before*** exception condition instruction complete

• Abort the instructions ***after*** ***excepting instruction***

• Save the state of the machine (esp. PC of the excepting condition instruction) to return back and continue from faulting instruction

• Start fetching instructions in memory where the ***exception handling routine*** instructions are kept

• This is known as implementing ***precise exceptions*** i.e. undo all instructions after the excepting instructions and restart from the excepting instruction



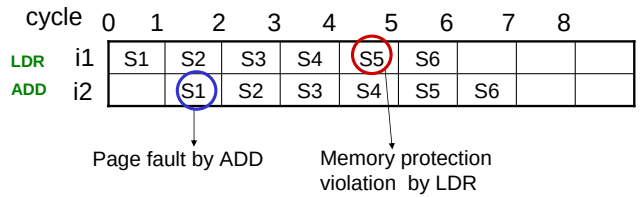
## Pipelining Complications

• Due to overlapping of instruction execution, multiple exceptions can occur in the same clock cycle

| Stage | Problem Exceptions Occurring |
|---|---|
| Fetch | Memory-protection violation, Page fault on instruction fetch, misaligned memory access |
| Decode | Undefined Instruction |
| Execute | Arithmetic exception |
| Memory Access | Memory-protection violation, Page fault on instruction fetch, misaligned memory access |



## Pipeline Complication: Example




The second instruction (ADD) produces an exception first, and then the first (LDR) instruction is restarted, then second instruction is executed twice!!!

## Solution to Multiple Exceptions in a Pipeline

Maintain exception vector for each instruction

- Some hardware logic handles this (similar to how NZP registers are maintained)
- Vector is nothing bit a n-bit register and each bit position indicating a stage of the pipeline
- To indicate exception set the appropriate bit position

When instruction enters the last stage of the pipeline, check the exception vector, and handle the exceptions in instruction order

- If the bit is set means that the instruction had faulted
- Abort all instructions following this instruction
- Save the state of the machine
- Branch to appropriate service routine



## Summary of Pipelining

- Improves performance
  - Improves runtime of program
    - ➢ Reducing the clock cycle time
      - – Increase Frequency (faster processing)
    - ➢ CPI = 1 (ideal)
  - Speedup = #number of pipe stages (ideal)
- However comes at price of greater CPI penalties
  - Data Hazard (Load-use delay)
  - Control Hazard (Branch/Jump delays)
  - Exceptions



## LC3 pipelined