Introduction to

Pi ion

COMP

Review (1/3)

- Datapath is the hardware that performs operations necessary to execute programs.

- Control instructs dat o next.

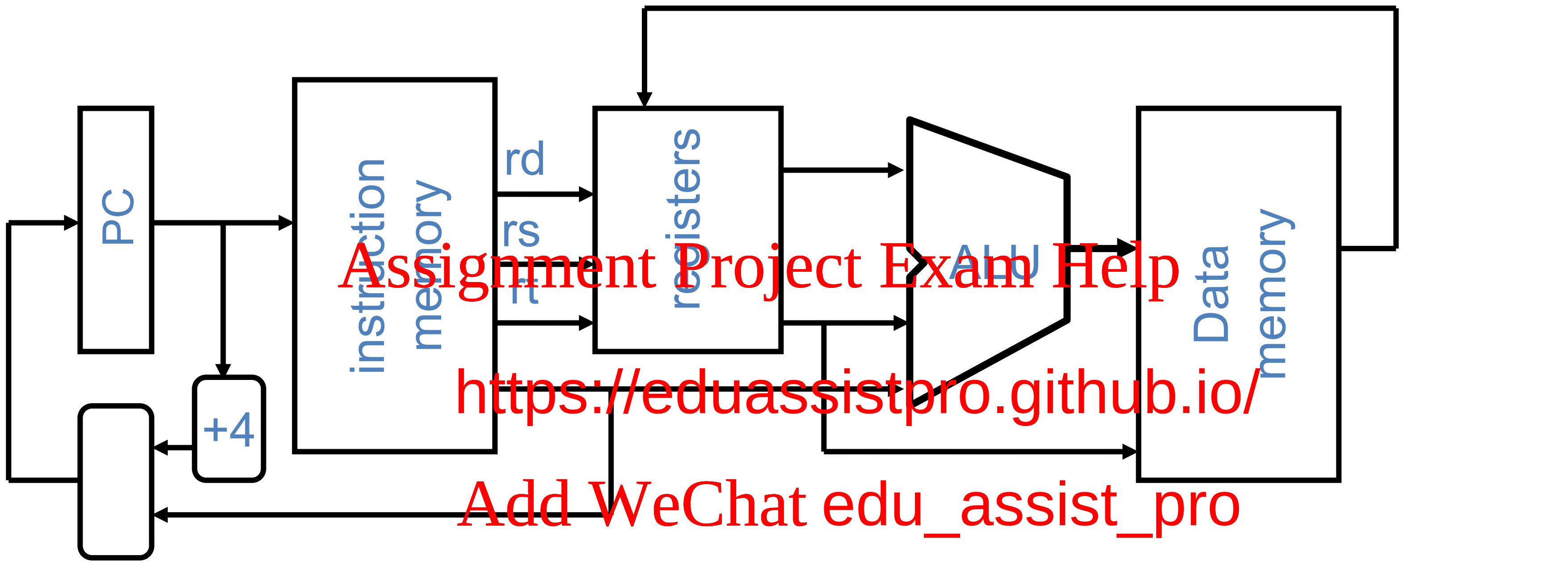
- Datapath needs:

  – access to storage (general purpos and memory)
  – computational ability (ALU)
  – helper hardware (local registers and PC)

Review (2/3)

- Five stages of datapath (executing an instruction):
  1. Instruction Fetch (Increment PC)
  2. Instruction s)
  3. ALU (Compu
  4. Memory Access
  5. Write to Registers
- ALL instructions must go through ALL five stages.

Review Datapath



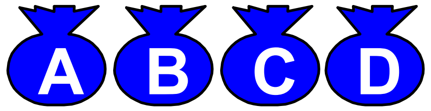
1. Instruction Fetch
2. Decode/ Register Read
3. Execute
4. Memory
5. Write Back

Outline

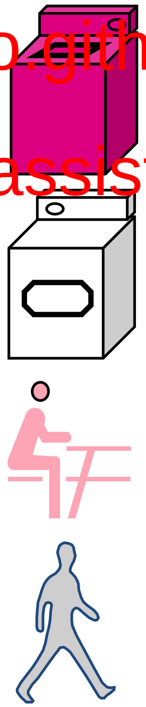
- Pipelining Analogy
- Pipelining Instruction Execution
- Hazards

With or without your laundry

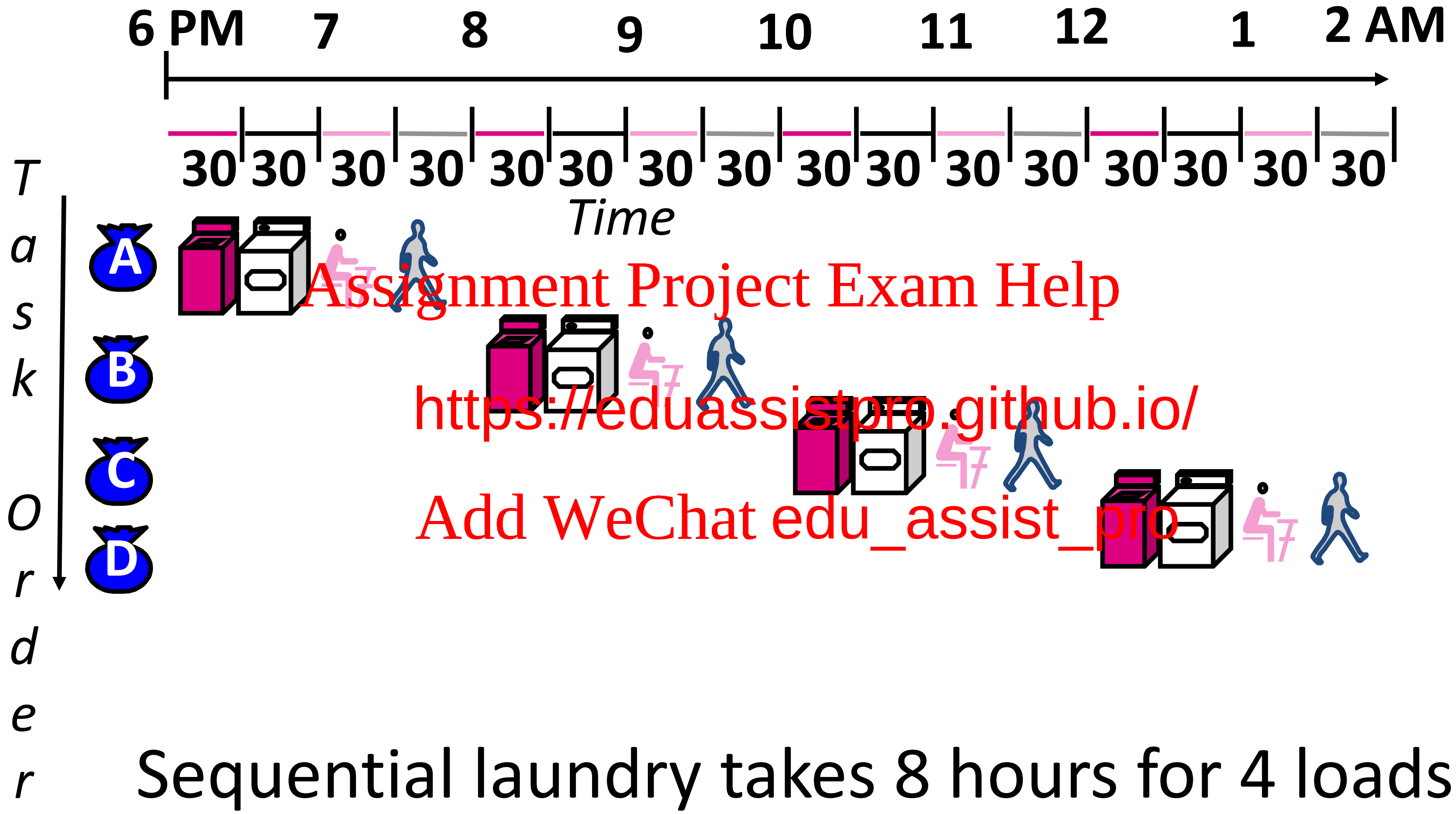
- Bono, The Edge, Adam Clayton, and Larry Mullen Jr., each have one load of clothes to wash, dry, fold, and put away



- **Washer takes 30 minutes**
- **Dryer takes 30 minutes**
- **"Folder" takes 30 minutes**
- **"Stasher" takes 30 minutes to put clothes into drawers**

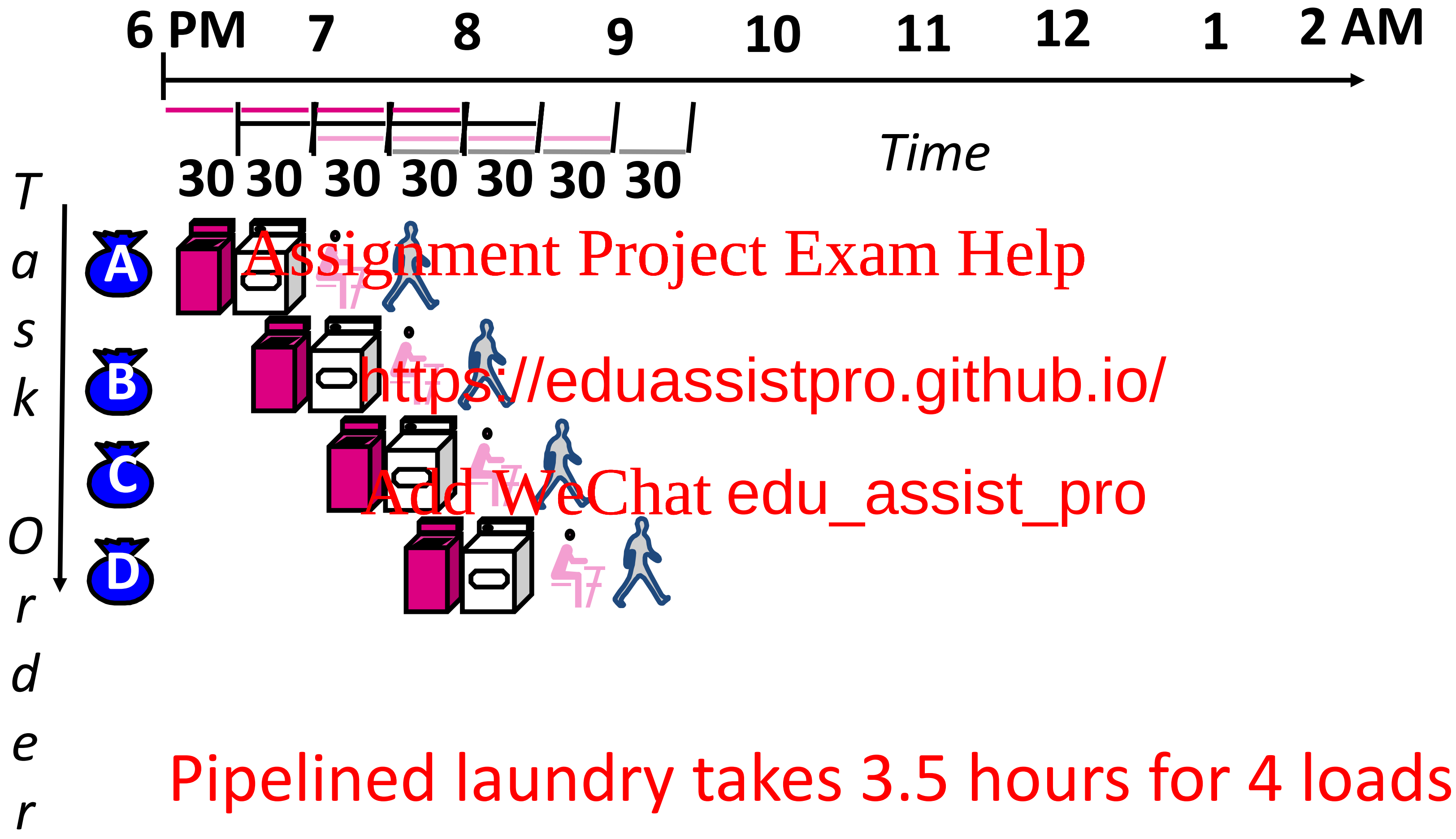


Sequential Laundry



Sequential laundry takes 8 hours for 4 loads

Pipelined Laundry

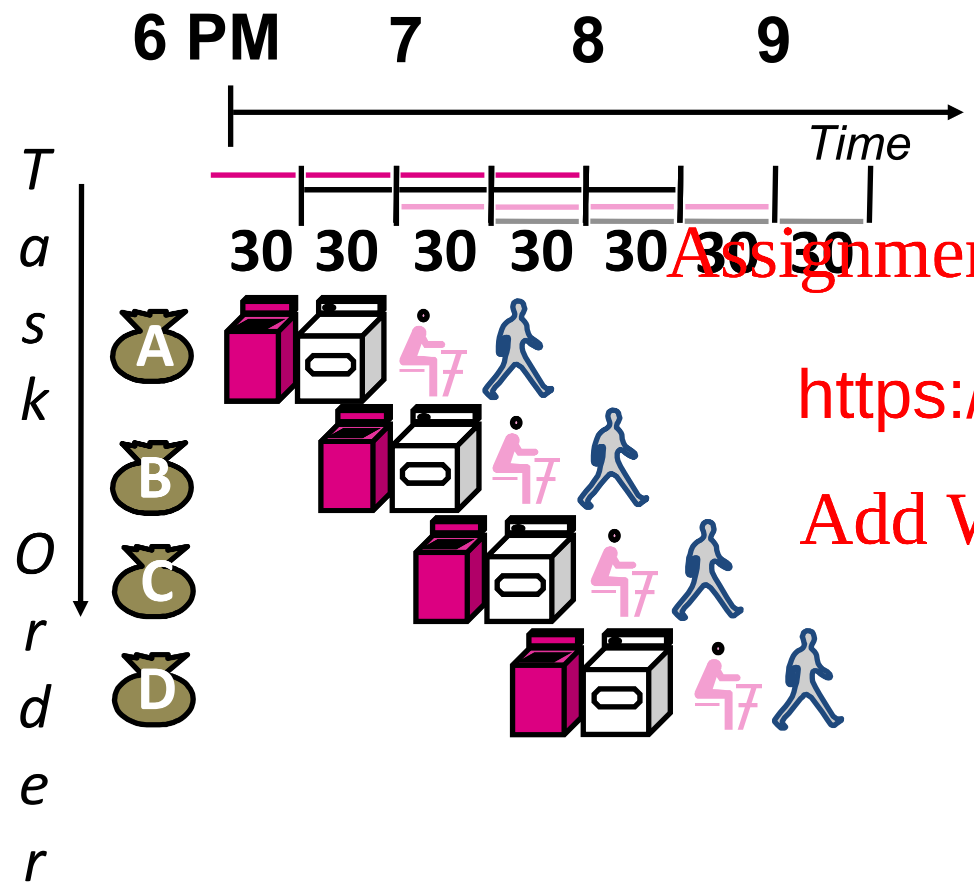


Pipelined laundry takes 3.5 hours for 4 loads!

General Definitions

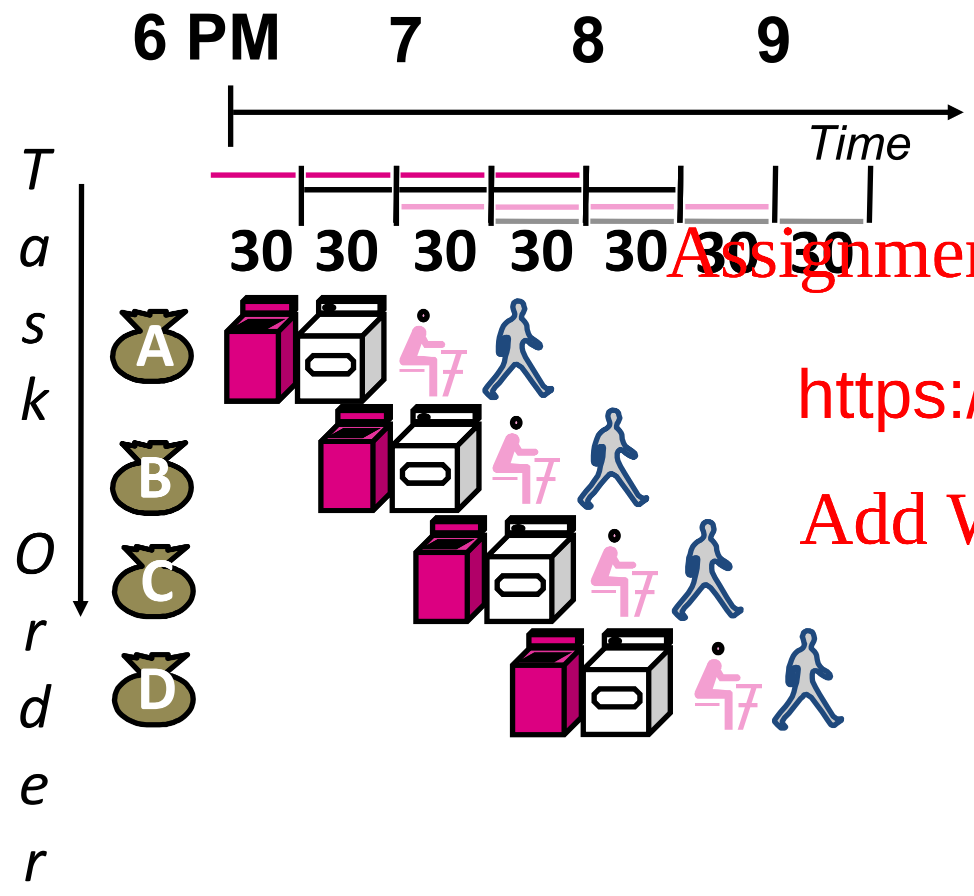
- **Latency**
  - Time to completely execute a certain task
    - For example, time to read a sector from disk is disk access time or disk latency
- **Throughput**
  - Amount of work that can be done [illegible]iod of time

Pipelining Lessons (1/2)



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- s operating simultaneously
- nt resources
- Pot dup = Number pipe stages
- Time to "fill" pipeline and time to "drain" it reduces speedup: 2.3X v. 4X in this example

Pipelining Lessons (2/2)



- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?

e limited by ***slowest***

ge

- Un ngths of pipe stages also reduces speedup

Steps in Executing MIPS

1) IFetch: Fetch Instruction, Increment PC

2) Decode: Instruction, Read Registers

3) Execute:

   Mem-ref: Calculat

   Arith-log: Perform Operation

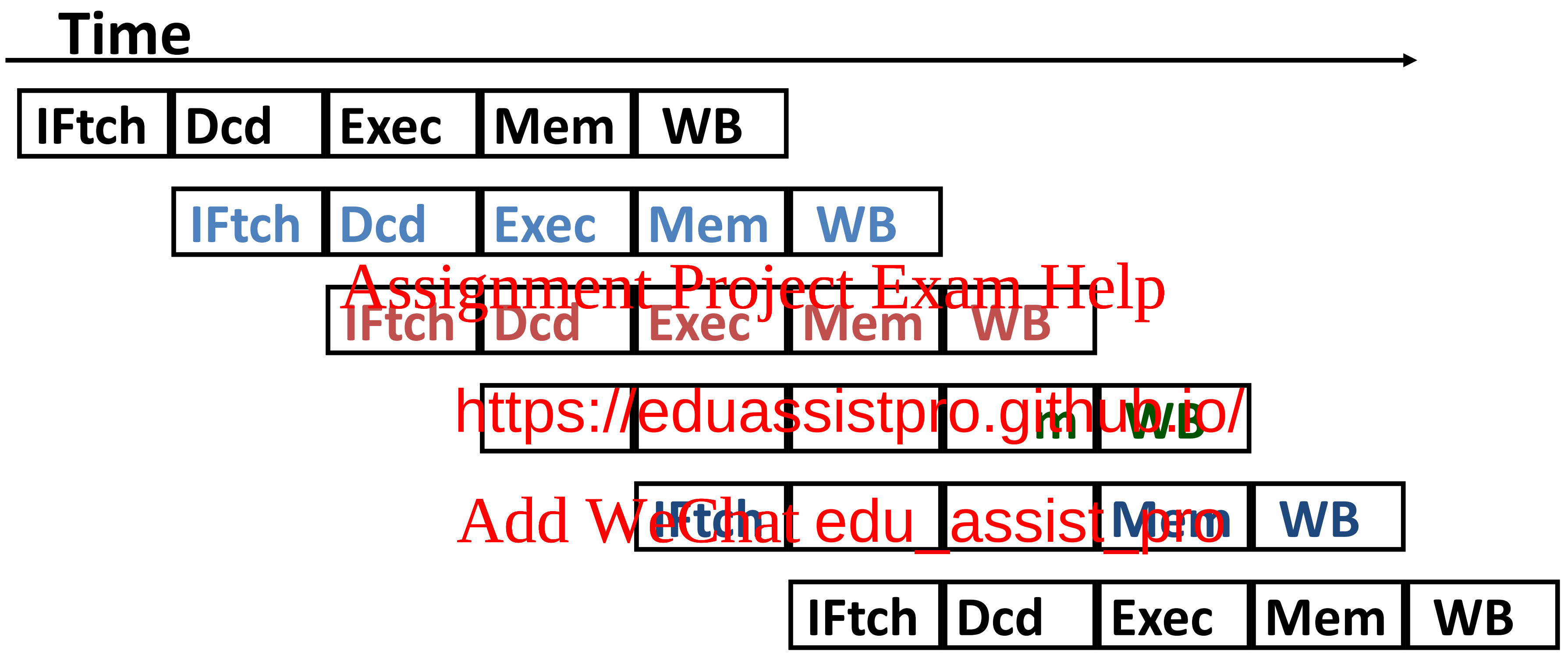
4) Memory:

   Load: Read Data from Memory

   Store: Write Data to Memory

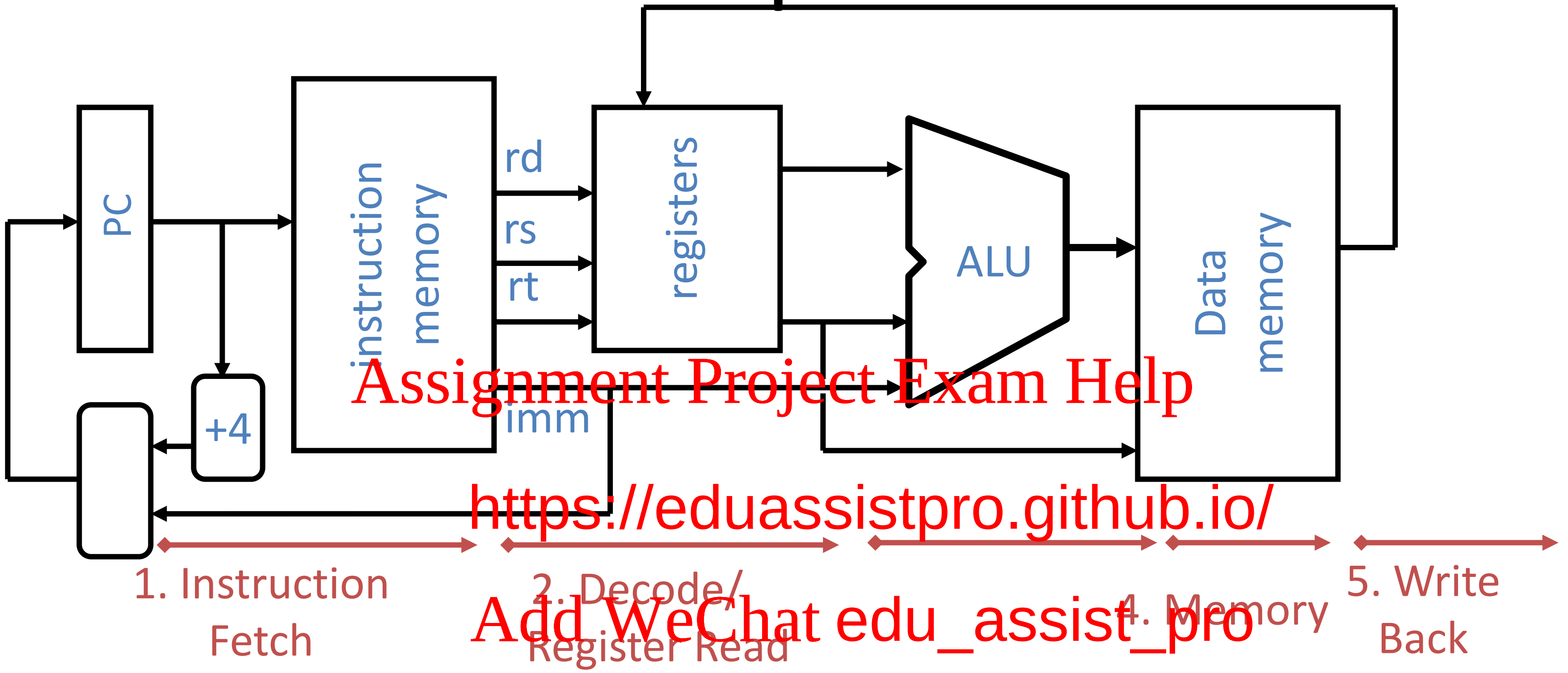
5) Write Back: Write Data to Register

Pipelined Execution Representation

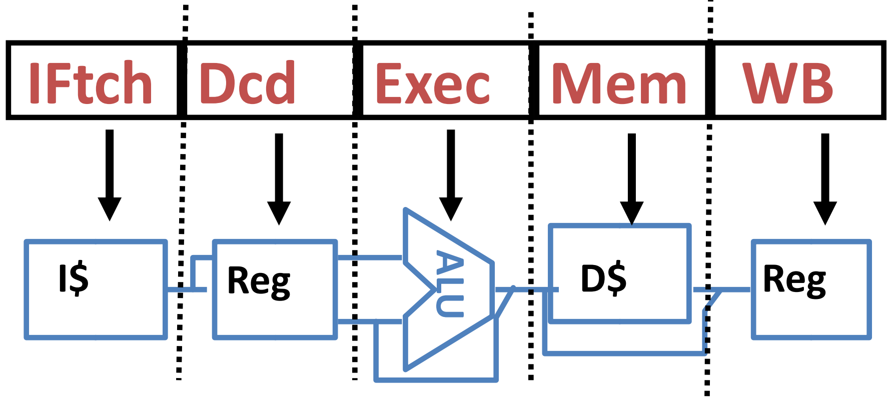


- Every instruction must take same number of steps, also called pipeline "***stages***", so some will go idle sometimes

Review: Datapath for MIPS

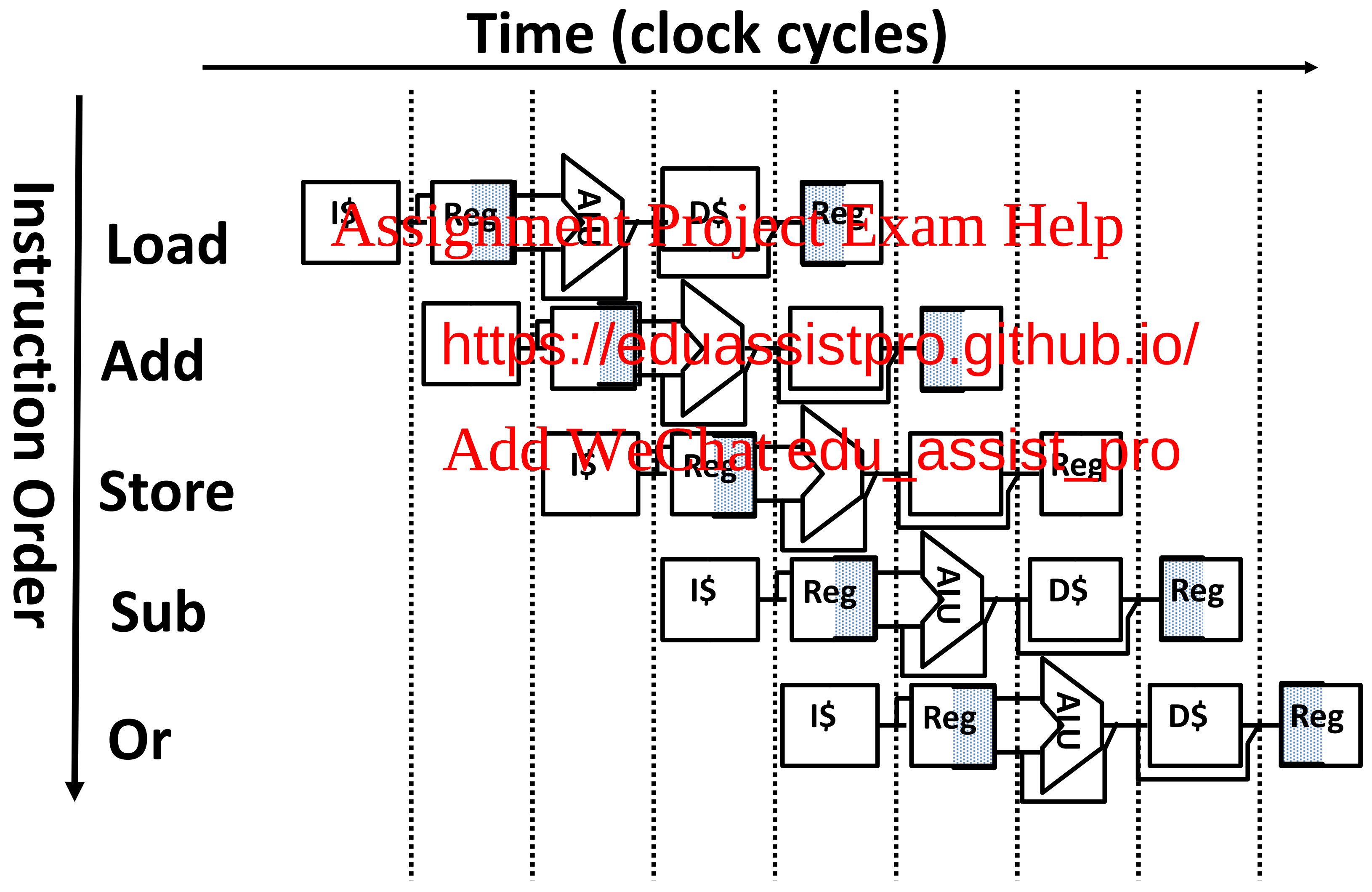


- Use datapath figure to represent pipeline



Graphical Pipeline Representation

(In Reg, right half highlight read, left half write)



Example

- Suppose 2 ns for memory access, 2 ns for ALU operation, and 1 ns for register file read or write; what is the ***instruction execution rate***?
- Non-pipelined Exec nd ADD:

  LW: IF + Read Reg + ALU + Memory +

  = 2 + 1 + 2 + 2 + 1 = 8 ns

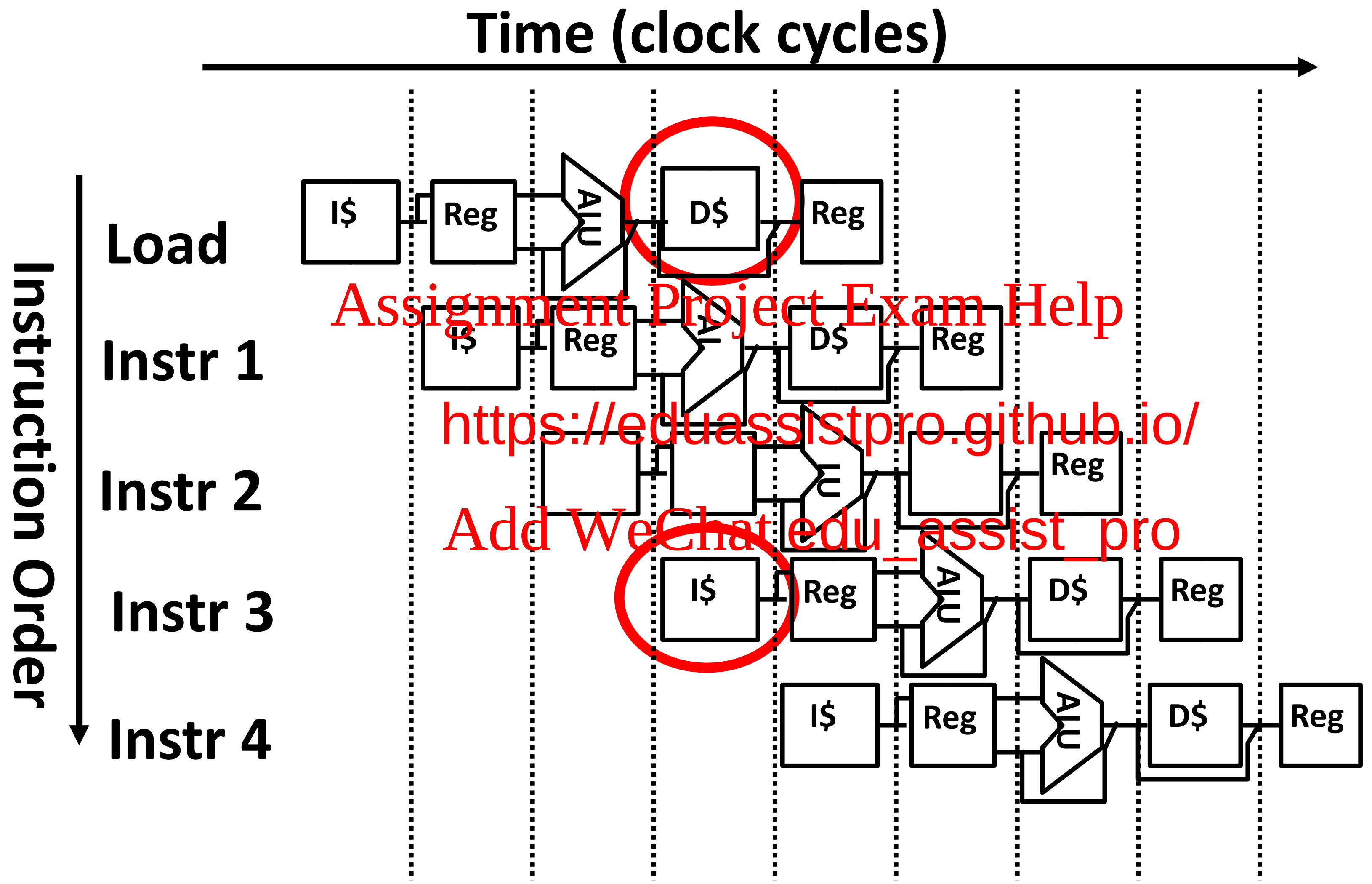
  ADD: IF + Read Reg + ALU + Write Reg
  = 2 + 1 + 2 + 1 = 6 ns
- Pipelined Execution:

  Max(IF,Read Reg,ALU,Memory,Write Reg) = 2 ns

Problems for Pipelined Processors

- There exist ***Hazards*** that prevent the next instruction from executing during its designated clock cycle
  - ***Structural hazards***: s combination of instructions
  - ***Control hazards***: Pipelining of bra her instructions can ***stall*** the pipeline until the hazard
  - ***Data hazards***: Instruction depends on result of prior instruction still in the pipeline

Structural Hazard #1: Single Memory (1/2)

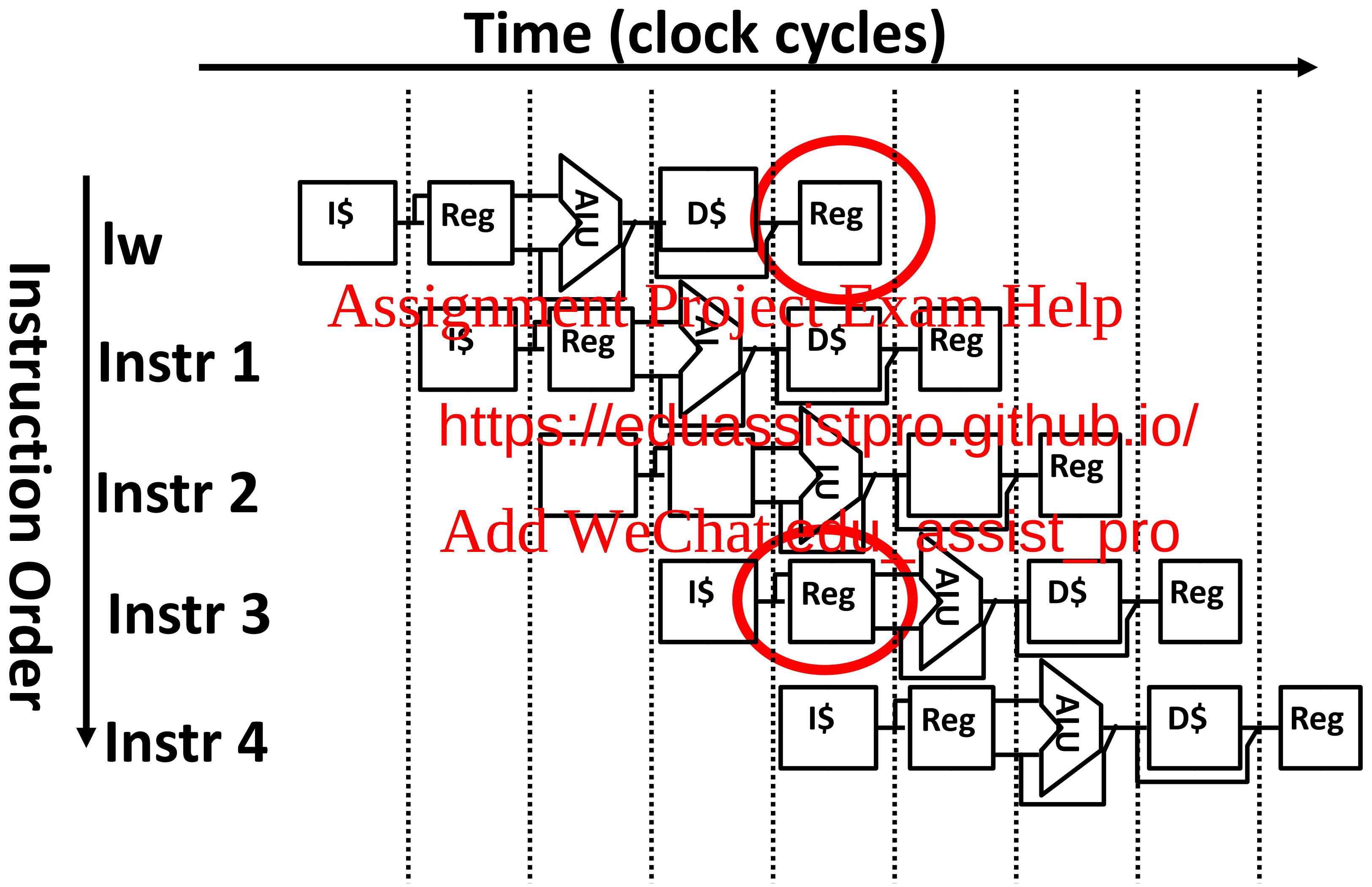


Read same memory twice in same clock cycle

Structural Hazard #1: Single Memory (2/2)

- Solution:
  - Infeasible and inefficient to create an independent second memory, but can simulate this by having two Level 1 Caches
  - Use an L1 Instructio Cache
  - Need more complex hardware to en both caches miss

Structural Hazard #2: Registers (1/2)

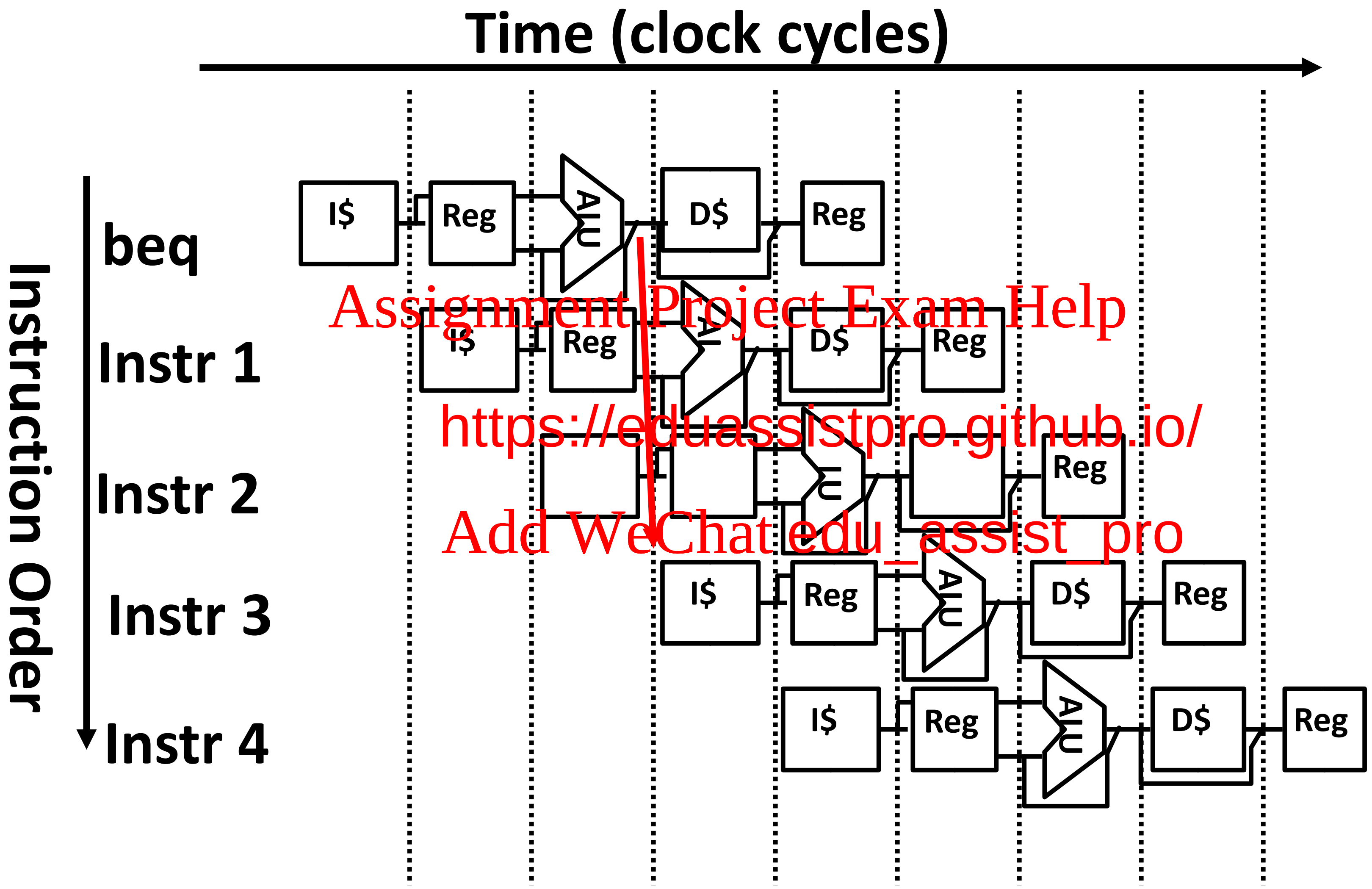


Can't read and write to registers simultaneously

Structural Hazard #2: Registers (2/2)

- Fact: Register access is ***VERY*** fast: takes less than half the time of ALU stage
- Solution: modify clo
  - always **Write** to Regi ch clock cycle
  - always **Read** from Registers durin **alf** of each clock cycle
  - **Result: can perform Read and Write during same clock cycle**

Control Hazard: Branching (1/7)



Where do we do the compare for the branch?

Control Hazard: Branching (2/7)

- We naively put branch decision-making hardware in ALU stage
  - therefore two more instructions after the branch will *always* be fetched, whether or not the branch is taken
- Consider: Desired fu [illegible] nch
  - if we do not take the branch, don’ [illegible] y time and continue executing normally
  - if we take the branch, don’t execute any instructions after the branch, just go to the desired label

Control Hazard: Branching (3/7)

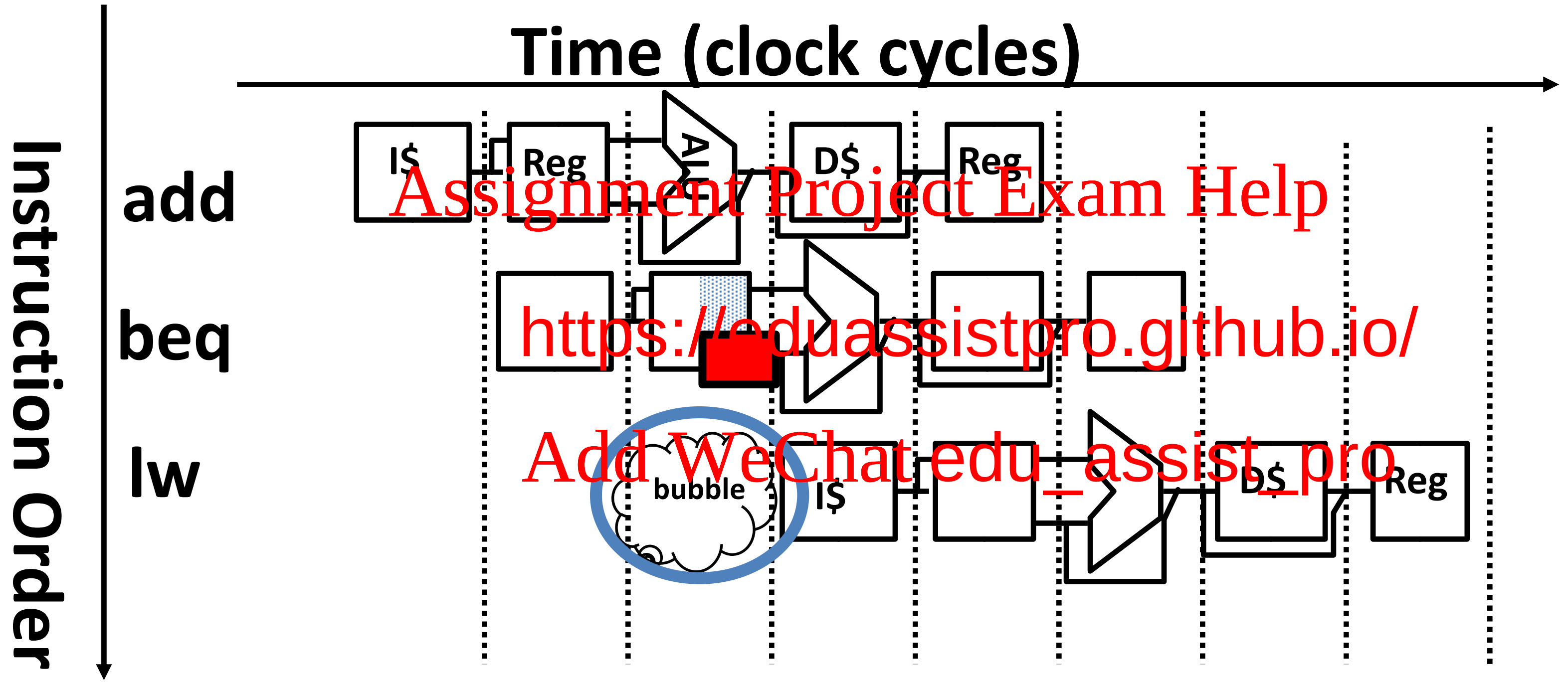
- Initial Solution: ***Stall*** until decision is made
  - insert “no-op” instructions that accomplish nothing, just take time
  - Drawback: branches ch (assuming comparator is put in ALU stage)

Control Hazard: Branching (4/7)

- Optimization #1:
  - Move asynchronous comparator up to Stage 2
  - As soon as instructio identifies is as a branch), immediately make a alue of the PC (if necessary)
  - Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
  - Side Note: This means that branches are idle in Stages 3, 4 and 5.

Control Hazard: Branching (5/7)

Insert a single no-op (bubble)



Impact: 2 clock cycles per branch instruction, slow

Control Hazard: Branching (6/7)

- Optimization #2: Redefine branches
  - Old definition: if we take the branch, none of the instructions after the branch get executed by accident

  - New definition: whe e branch, the single instruction immediately following h gets executed

  - This is called the ***branch-delay slot***

Control Hazard: Branching (7/7)

- Notes on ***Branch-Delay Slot***
  - – Worst-Case Scenario: can always put a no-op in the branch-delay slot
  - – Better Case: can find ing the branch which can be placed in the bra affecting flow of the program
    - ***Re-ordering instructions*** is a common method of speeding up programs
    - Compiler must be very smart in order to find instructions to do this
    - **Usually can find such an instruction at least 50% of the time**
    - Jumps also have a delay slot!

Example: Nondelayed vs. Delayed Branch

Nondelayed Branch

```
or    $8, $9 ,$10

add $1 ,$2,$3

sub $4, $5,$6

beq $1, $4, Exit

xor $10, $1,$11

Exit:
```

What can we stick in the branch delay slot?

Delayed Branch

```
add $1 ,$2,$3

sub $4, $5,$6

beq $1, $4, Exit

or  $8, $9 ,$10

xor $10, $1,$11

Exit:
```

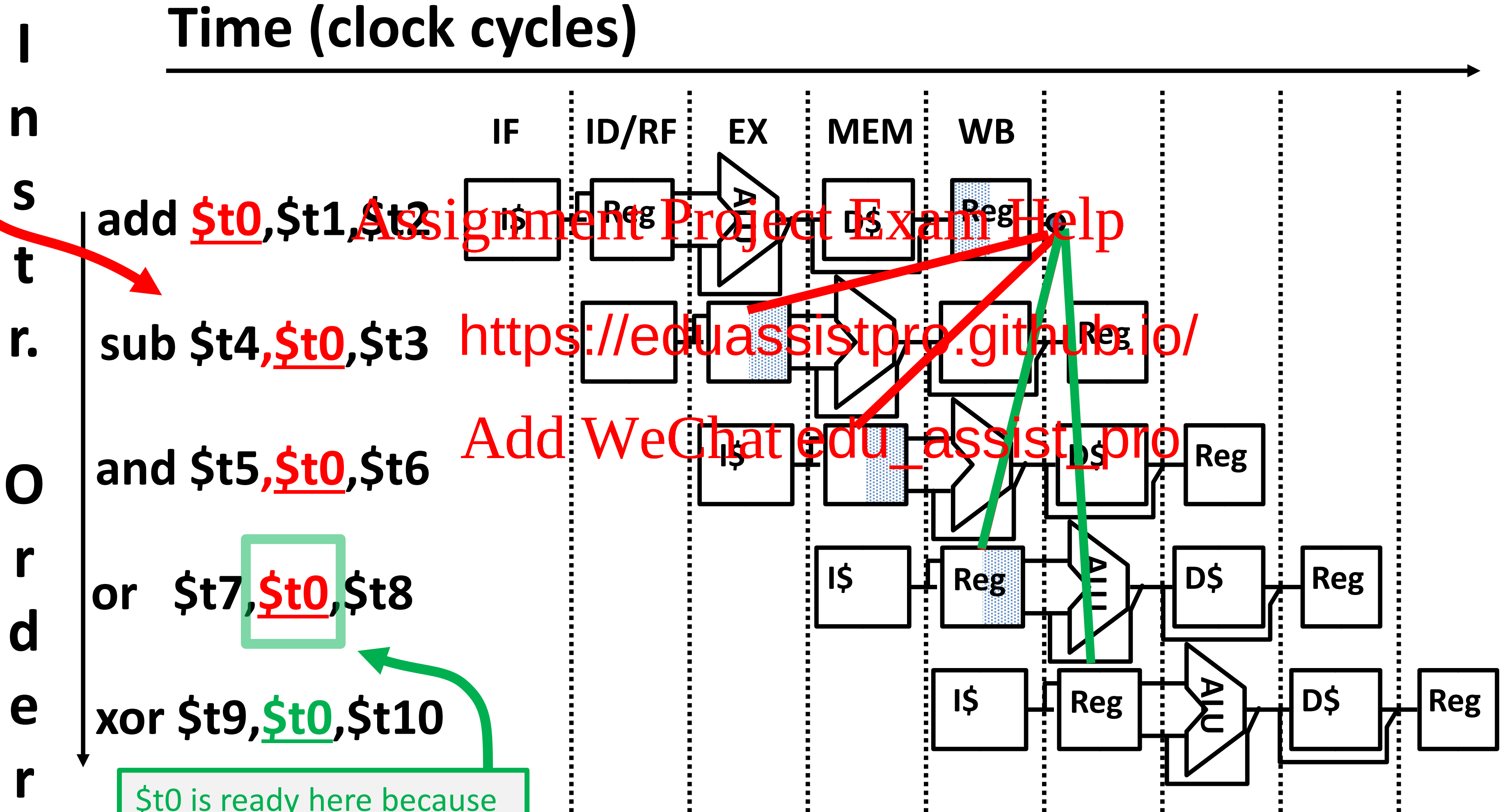
Simulating Hazards

- MARS can simulate delayed branches

Sub will try to read $t0 before the result is written!

Data Hazards

Dependencies *backwards in time* are hazards



$t0 is ready here because we wrote back in first half of clock cycle

Data Hazard Solution: Forwarding

Forward result from one stage to another

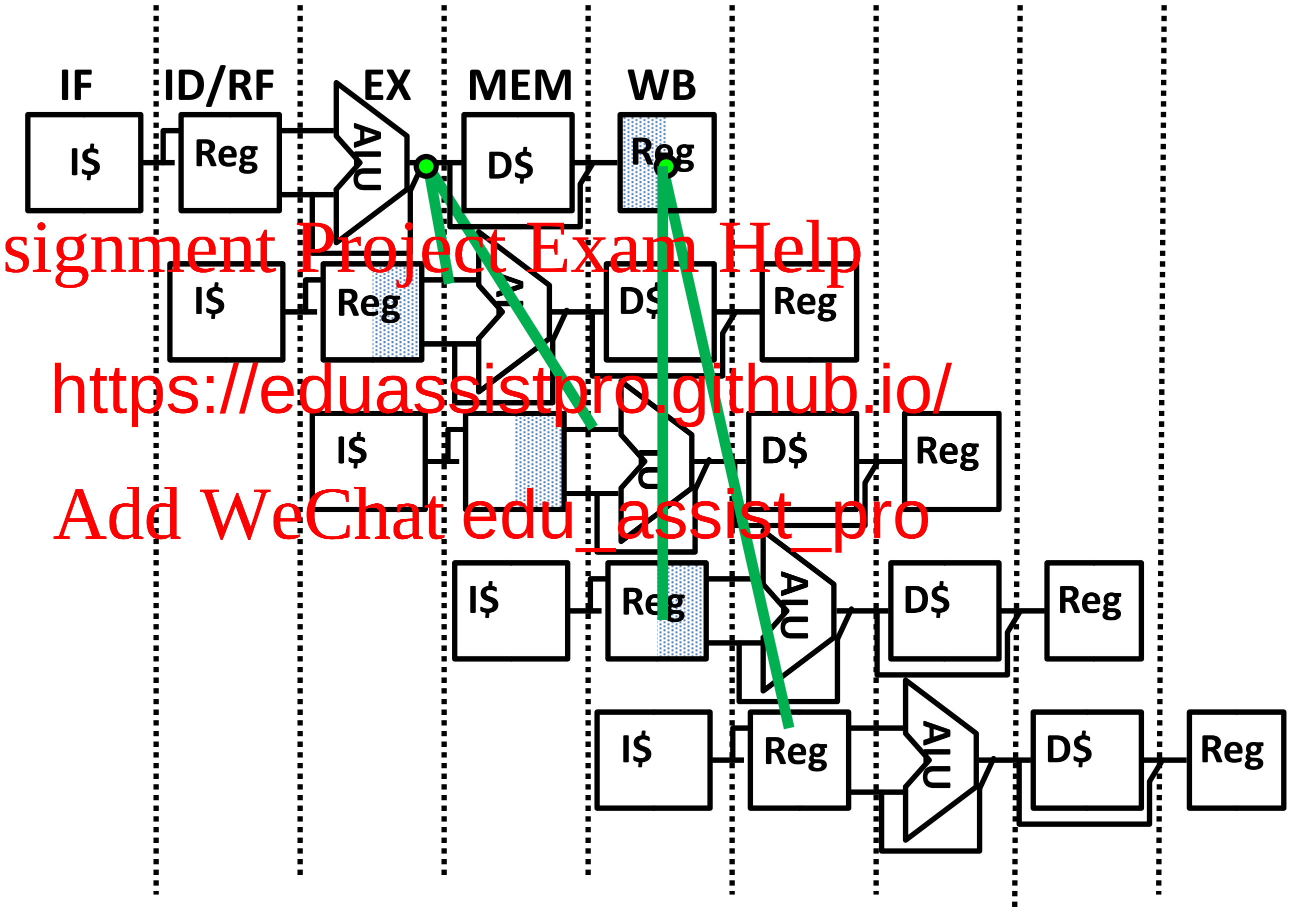
add $t0,$t1,$t2

sub $t4,$t0,$t3

and $t5,$t0,$t6

or $t7,$t0,$t8

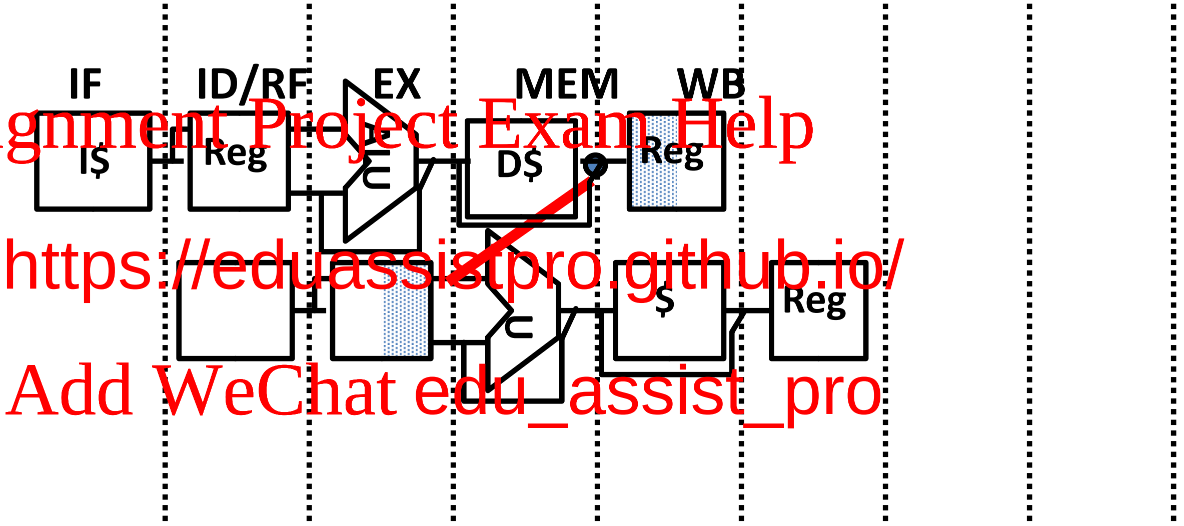
xor $t9,$t0,$t10



"or" hazard solved by register hardware

Data Hazard: Loads (1/4)

- Dependencies backwards in time are hazards



lw $t0,0($t1)

sub $t3,$t0,$

- Can't solve with forwarding
- Must ***stall*** instruction dependent on load, then forward (more hardware)

Data Hazard: Loads (2/4)

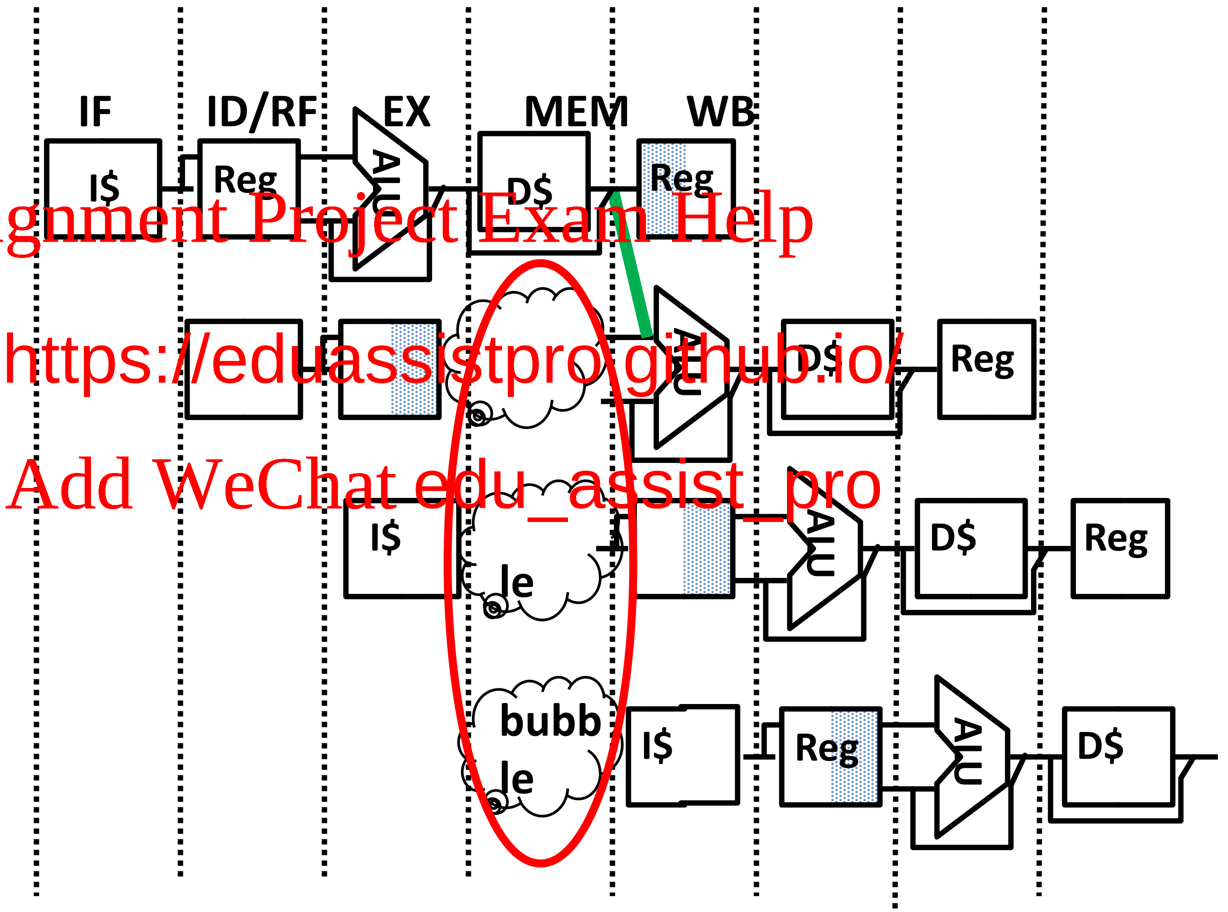
- Hardware must ***stall*** pipeline
- Also called "***interlock***"

lw $t0, 0($t1)

sub $t3,$t0,$t2

and $t5,$t0,$t4

or $t7,$t0,$t6

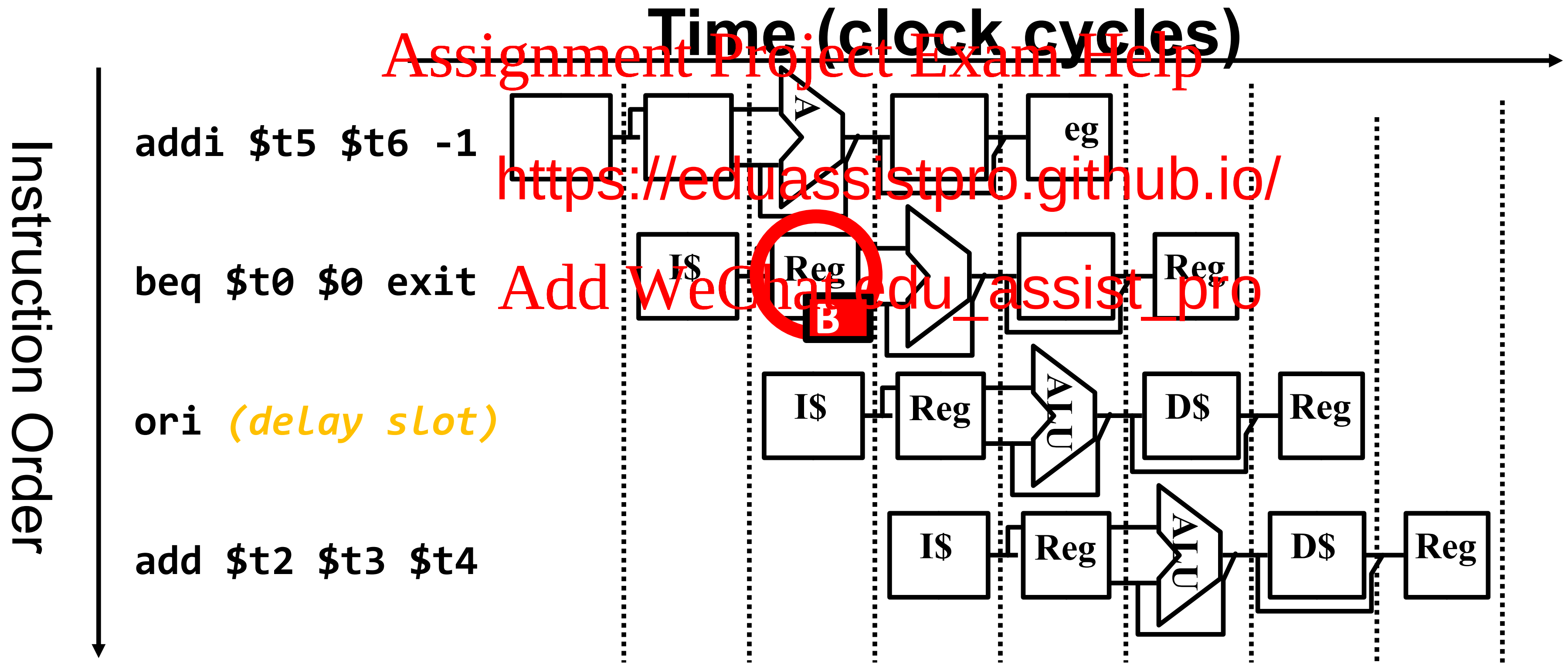


Data Hazard: Loads (3/4)

- Instruction slot after a load is called ***load delay slot***
- If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle.
- If compiler puts an un that slot, then no stall
- Letting the hardware stall the inst the delay slot is equivalent to putting a nop in the
- *Alternatively: could redefine instruction semantics to introduce a delay slot, i.e., after lw (or lb or lbu) instruction, the data isn't actually available in the destination register until an extra instruction later*

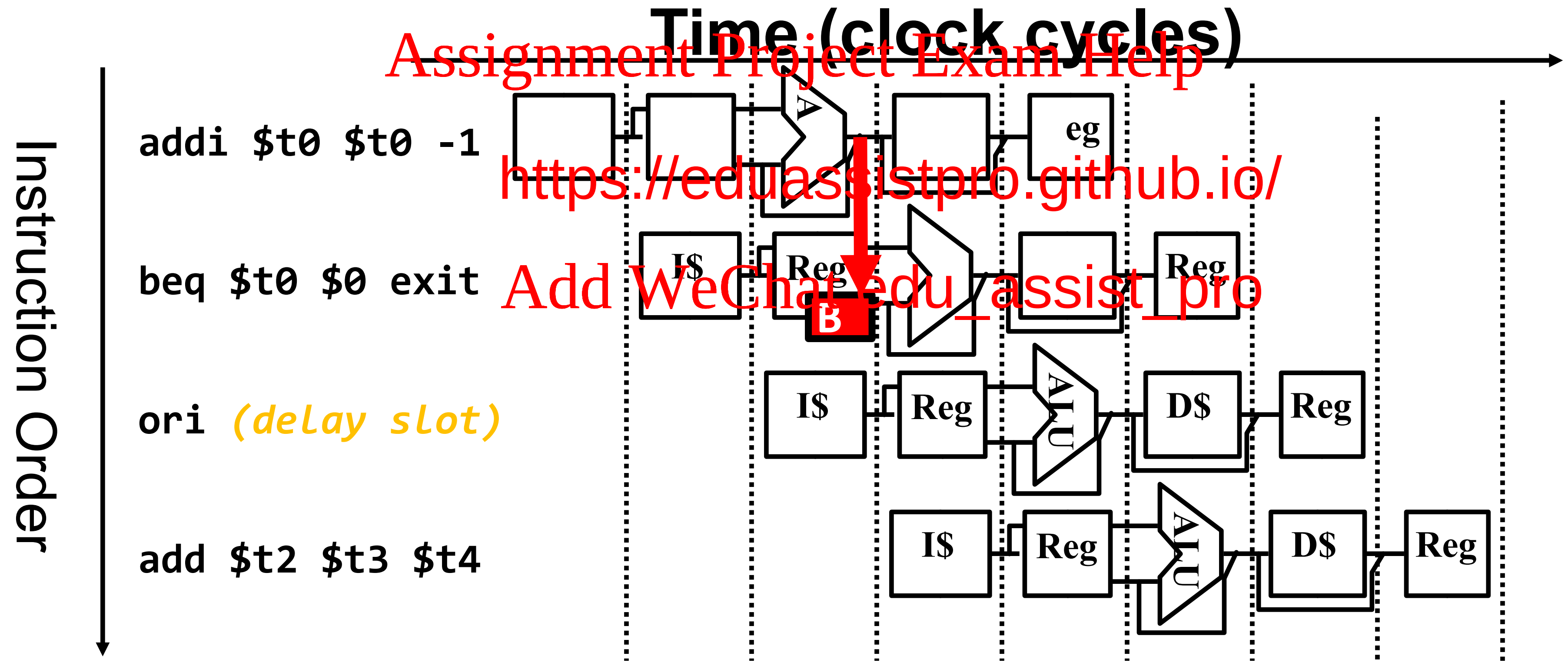
Recall: Control Hazard

Decision to branch made during instruction decode, registers can be read in the 2nd half of the cycle. Comparison of values very fast with simple XNOR + AND.



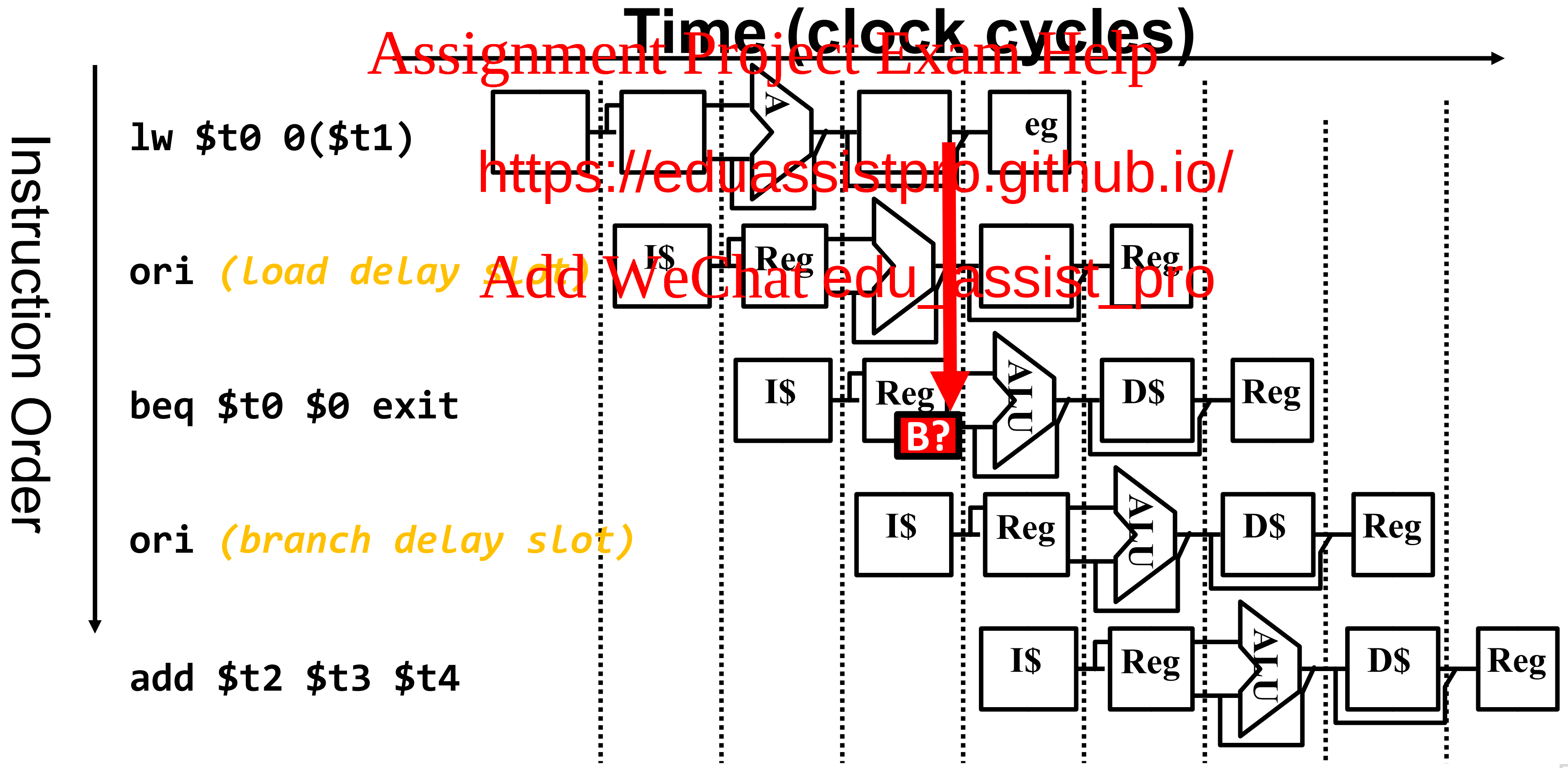
Data-Control Hazard?

Decision to branch might also need a result from the ALU.
Can we assume that the branch comparison is fast enough to asynchronously access the ALU result in the ID stage?



Data-Control Hazard?

Even with the load delay slot, we may need to asynchronously access the load from memory to make the branch decision.



Optimization (1/3)

- Now that we know what is fast and what is slow, how do we write fast programs?
  - As long as we avoid ding pipeline stalls, we maximize instructio
- First, simplest technique: maint y
  - Stalling a cycle or two for a control/data hazard is nothing compared to stalling hundreds of cycles for a cache miss!

Optimization (2/3)

- Instruction reordering:
  - Be aware of delay slots, reorder instructions to put a useful yet unrelated instruction in a delay slot.
  - This is a pretty tedio even assemblers do a good job of doing the dirty work f

Question

- **Assume 1 instr/clock, delayed branch, 5 stage pipeline, forwarding, interlock on unresolved load hazards (after 10^3 loops, so pipeline full)**

```
Loop:      lw
           add
           sw     $t0, 0($
           addiu  $s1, $s1,
           bne    $s1, $zero, Loop
           nop
```

- **How many pipeline stages (clock cycles) per loop iteration to execute this code?**

Answer

- **Assume 1 instr/clock, delayed branch, 5 stage pipeline, forwarding, interlock on unresolved load hazards (after 10^3 loops, so pipeline full)**

es stall)

```
Loop:     1 lw
          3 add
          4 sw     $t0, 0($
          5 addiu $s1, $s1,
          6 bne    $s1, $s3, Loop
          7 nop
```

(delayed branch means we execute this instruction)

- **How many pipeline stages (clock cycles) per loop iteration to execute this code?**

Optimization Question

- Instruction Reordering Example
  - The loop takes 7 clock cycles per iteration. Can you improve it?

```
Loop: l                    )
      n
      addu  $t0,          s2
      sw    $t0, 0($s1)
      addiu $s1, $s1, -4
      bne   $s1, $s3, Loop
      nop
```

Intruction Reordering Solution

- Reordering can reduce the number of cycles to 5 per iteration:

```
Loop: l
      a
      addu $t0,        2
      bne $s1, $s3, loop
      sw $t0, 4($s1)
```

Optimization (3/3): Loop Unrolling

- Branches are difficult, just avoid them if possible.
- For a loop with a fixed number of iterations, write the assembly code by ju of the loop out repeatedly rather th l branches.
- Tradeoff: ***more total code*** but ***fe uctions*** executed overall and fewer branch instructions → means faster execution time.
- Can also partially unroll large loops!

Loop Unrolling Example

```
       addi $s0, $0, $0      # $s0 = 0
loop:  lw   $t0, 0($s1)
       add  $s2, $s2, $t0
       addi $s1, $s
       addi $s0, $s0, 1
       slt  $t1, $s0, 4      # if $s0 < 4
       bne  $t1, $0, loop    # then loop
```

If this loop were longer, it would be useful to eliminate the counter s0 and instead keep track et end address s1+4n and eliminate ctions in the loop

What does this code do

Loop Unrolling Example

```
lw $t0, 0($s1)
add $s2, $s2, $t0
lw $t0, 4($s1)
add $s2, $s2, $t0
lw $t0, 8($s1)
add $s2, $s2, $t0
lw $t0, 12($s1)
add $s2, $s2, $t0
```

Loop Unrolling Example

```
lw $t0, 0($s1)
lw $t1, 4($s1)
lw $t2, 8($s1)
lw $t3, 12($s1)
add $s2, $s2, $t0
add $s2, $s2, $t1
add $s2, $s2, $t2
add $s2, $s2, $t3
```

rrange to
egisters

Loop Unrolling Example

```
lw $t0, 0($s1)
lw $t1, 4($s1)
add $s2, $s2, $t0
add $s2, $s2, $t1
lw $t0, 8($s1)
lw $t1, 12($s1)
add $s2, $s2, $t0
add $s2, $s2, $t1
```

rrange to
egisters

Branch Prediction

- Modern processors with long pipelines also use a technique called *branch prediction*.
- As soon as any bran hed, make an instantaneous guess branch will be taken or not and keep executing assumin ssed right.
- If we guess wrong, undo everything that we shouldn't have done and start over at the correct place.

Branch Prediction

- A naïve branch prediction algorithm:
  - If the branch target is backward, guess that it will be taken. If the target is forward, guess that it will not be taken.
  - What's the rationale

- More elaborate algorithms actu how often each branch instruction is taken and modify their behavior – called *dynamic branch prediction*.

The Big Picture

- Although the compiler generally relies on the hardware to resolve hazards and thereby ensure correct execution, the compiler must unde to achieve the best performance. Other talls will reduce the performance of the compiled co

Questions

- The book uses the excellent "washing and drying clothes" analogy to explain pipelines and the three common hazards. Describe another an lines are useful and explain what the thr at domain. Aim for an analogy as far from clothes (and rs) as possible -- think outside the box.
- What requirements did the original MIPS processor place on compilers to avoid the need for hardware to stall the pipeline?

Questions

A. Thanks to pipelining, I have reduced the time it took me to wash my shirt.

B. Longer pip win (since less work per stage & a fas).

C. We can rely on compilers to help us avoid data hazards by reordering instructions.

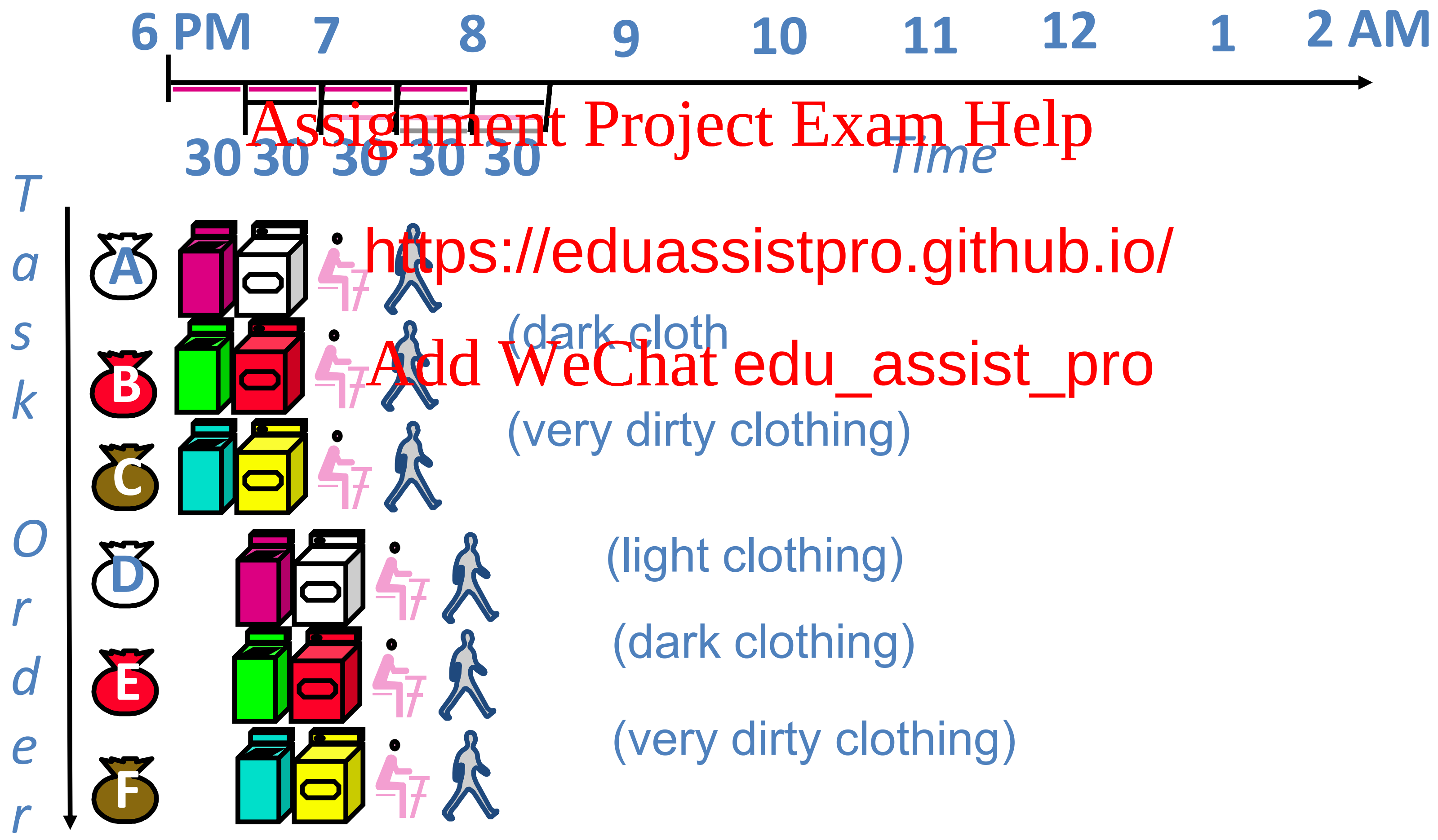
Exception Handling

- If the datapath and control aren't already complex enough, what about exceptions?
- On an exception, ne *tely* to the exception handler
  - Exceptions really must be precise, ust arbitrarily redefine the behavior to add delay slots.
- ***Definitely an advanced topic, beyond the scope of this course…***

Superscalar Laundry: Parallel per stage

More resources,

HW to match mix of parallel tasks?



Things to Remember (1/2)

- Optimal Pipeline
  - Each stage is executing part of an instruction each clock cycle.
  - One instruction finishes during each clock cycle.
  - On average, execute
- What makes this work?
  - Similarities between instructions allow us to use same stages for all instructions (generally).
  - Each stage takes about the same amount of time as all others: little wasted time.

Things to Remember (2/2)

- Pipelining is a BIG IDEA
  - widely used concept
- What makes it less than perfect?
  - Structural hazards: [illegible] ne cache?

    ⇒ Need more HW resources
  - Control hazards: need to worry a [illegible] h instructions?

    ⇒ Delayed branch
  - Data hazards: an instruction depends on a previous instruction?
- Advanced techniques: branch prediction, out of order execution, superscalar

Review and More Information

- Textbook Section 4.5 and 4.6
- Hazards in Sections 4.7 and 4.8

Extra Question

- Implement the memory copy function

```
    memcopy( int[] A, int[] B, int n ) {
        for ( int i = 0; i < n; i++ ) A[i] = B[i];
    }
```

  - Assume a MIPS machin [illegible] ck cycle, delayed branching, a 5 stage pipeline, forwarding, and interlo [illegible] olved load hazards
  - Respect register conventions
  - Use only true assembly language
  - Use careful instruction ordering to make a loop that takes the shortest possible number of cycles to complete

Extra Question, Part 2

- Partially unroll the memory copy function

```
blockcopy( int[] A, int[] B, int n ) {
    for ( int i = 0; i < 4*n; i+=4 ) {
        A[i] = B[i];
        A[i+1] = B[i+1]
        A[i+2] = B[i+2]
        A[i+3] = B[i+3];
    }
}
```

  - Same assumptions as before
  - Respect register conventions and only use true MIPS
  - Use careful instruction ordering to produce a loop that takes the shortest possible number of cycles to complete

Questions

- How much faster is blockcopy than memcopy?
  - Can you make memcopy do 1 word in 5 cycles?
  - Can you make block cycles?
  - If yes, 1.82 times fas