UNIVERSITY OF TEXAS ARLINGTON

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 18: More Processor Pipeline, Other Parallelism, and Debugging with GDB

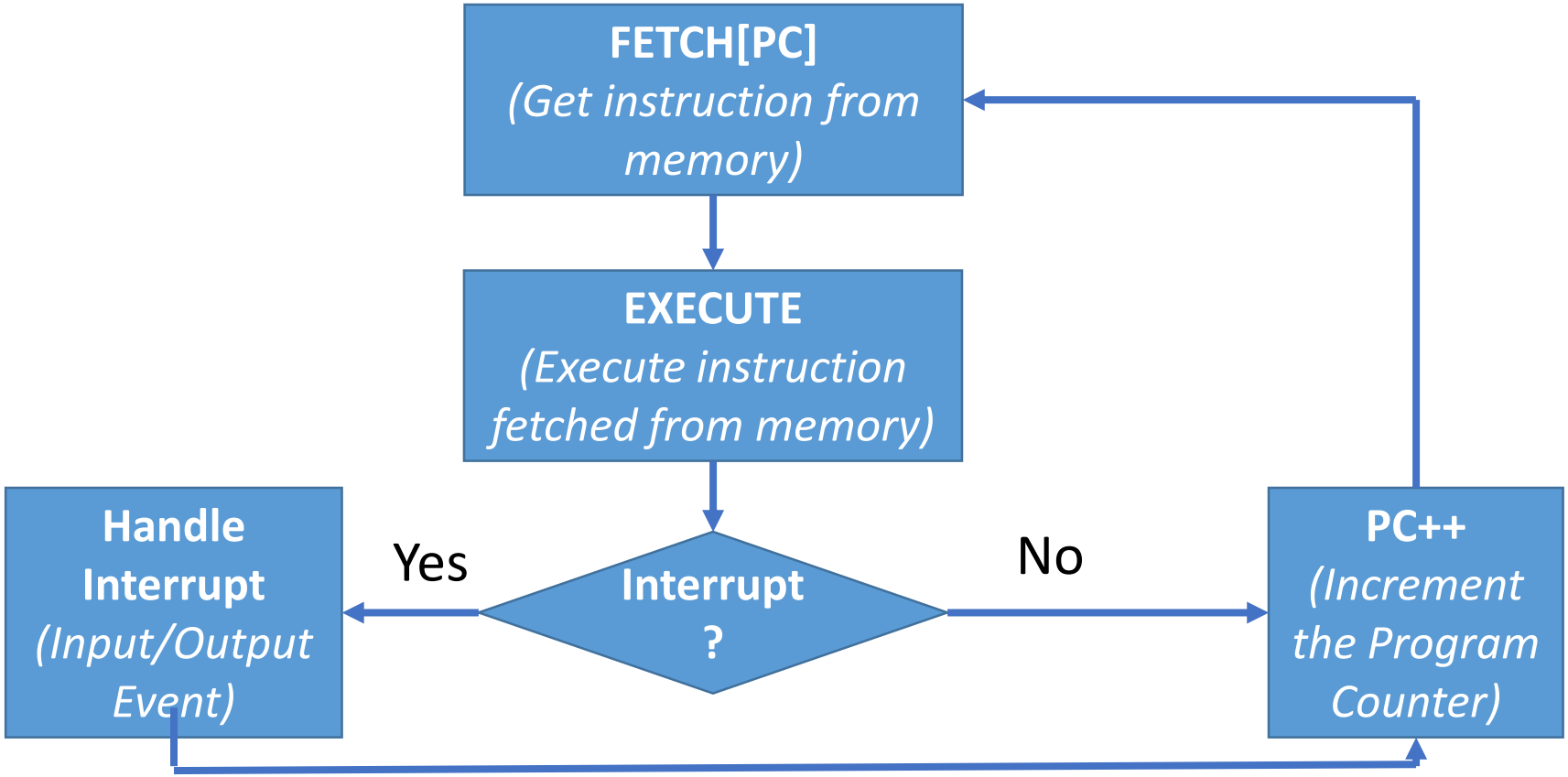
Taylor Johnson

Announcements and Outline

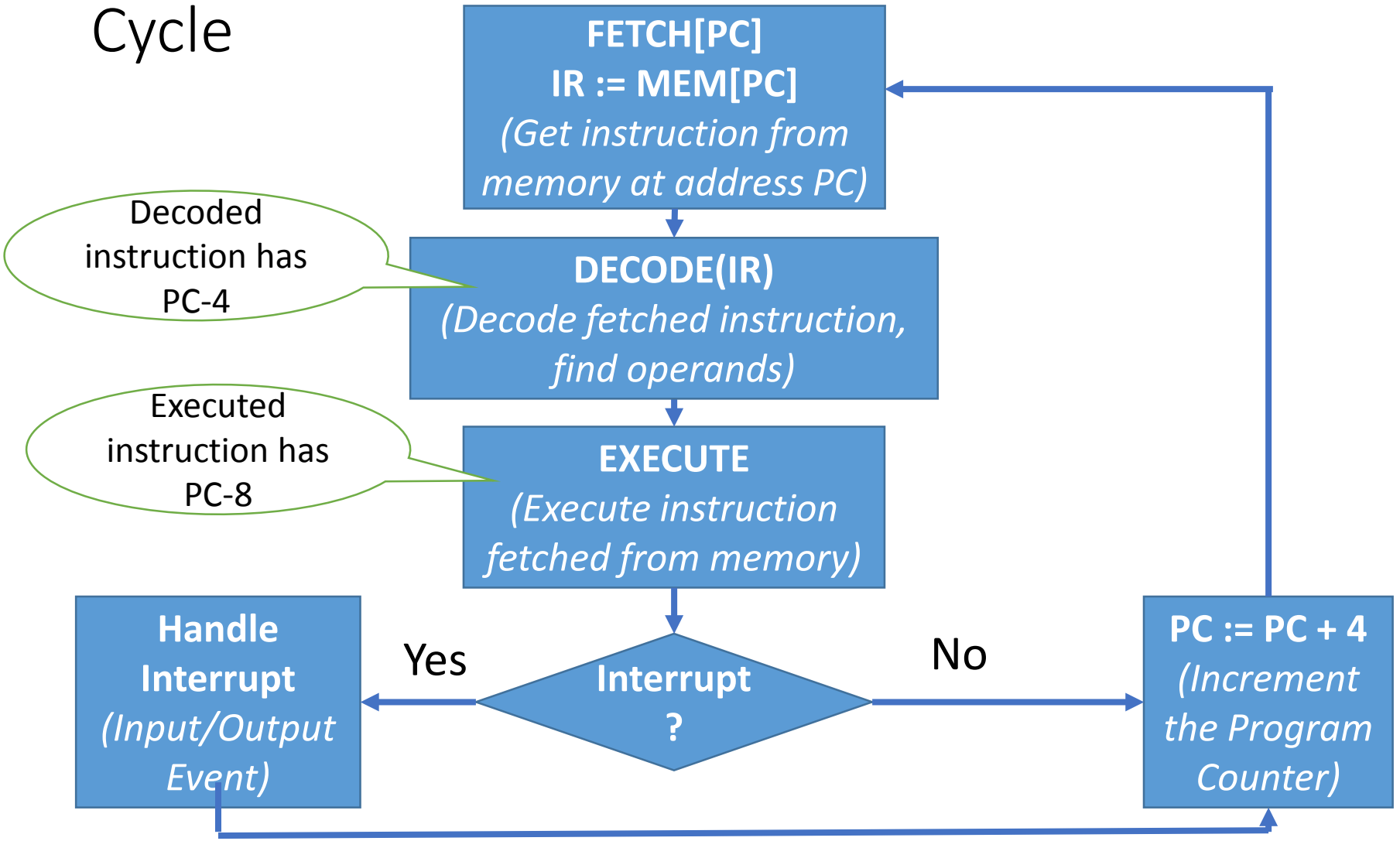
- Programming assignment 1 assigned soon

- More Pipelining
- More Running ARM assembly programs with QEMU and debugging with gdb
  - Debugging a basic procedure and looking at the stack
  - Debugging a recursive procedure and looking at the stack

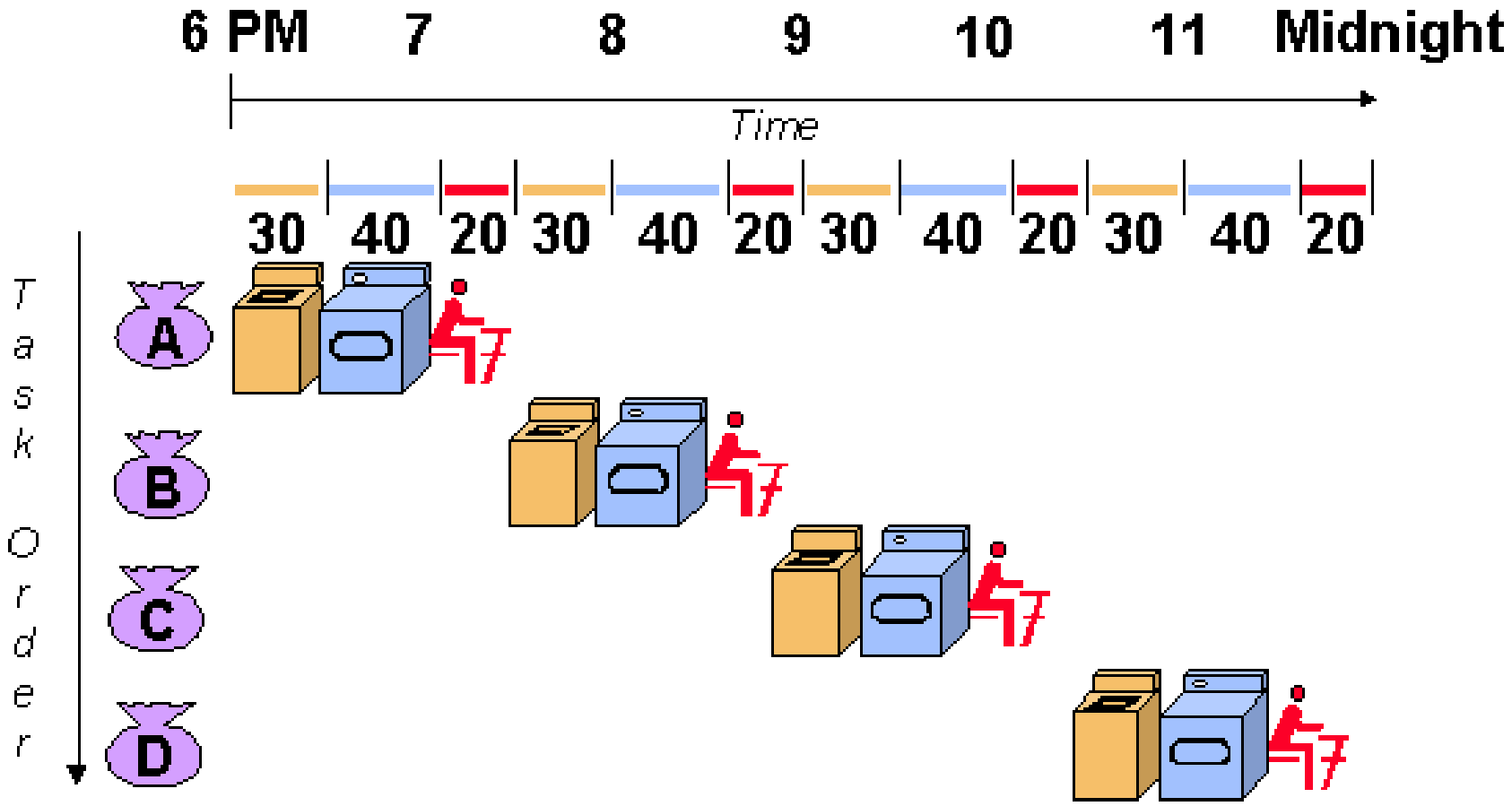
Review: Abstract Processor Execution Cycle



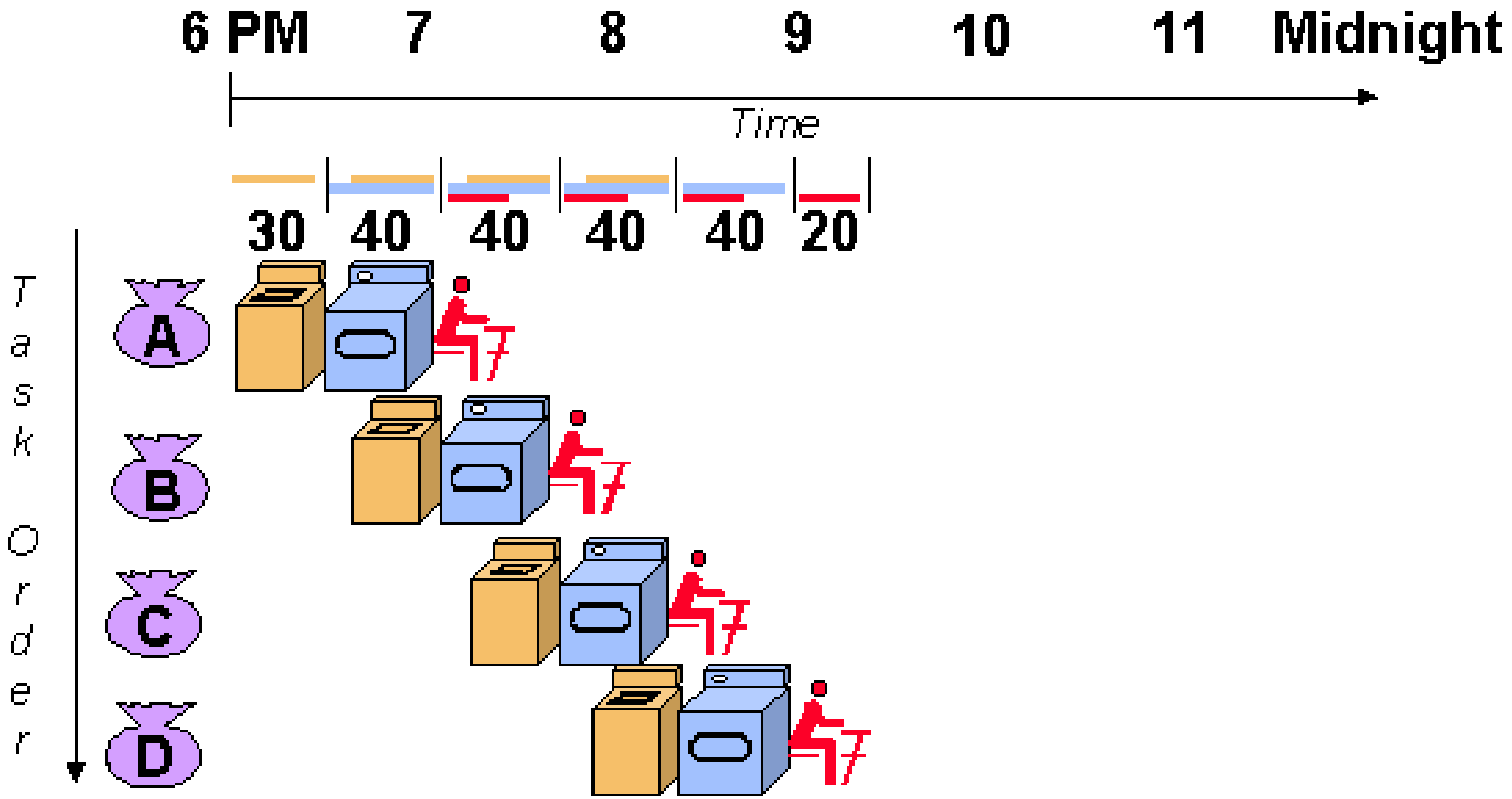
ARM 3-Stage Pipeline Processor Execution Cycle



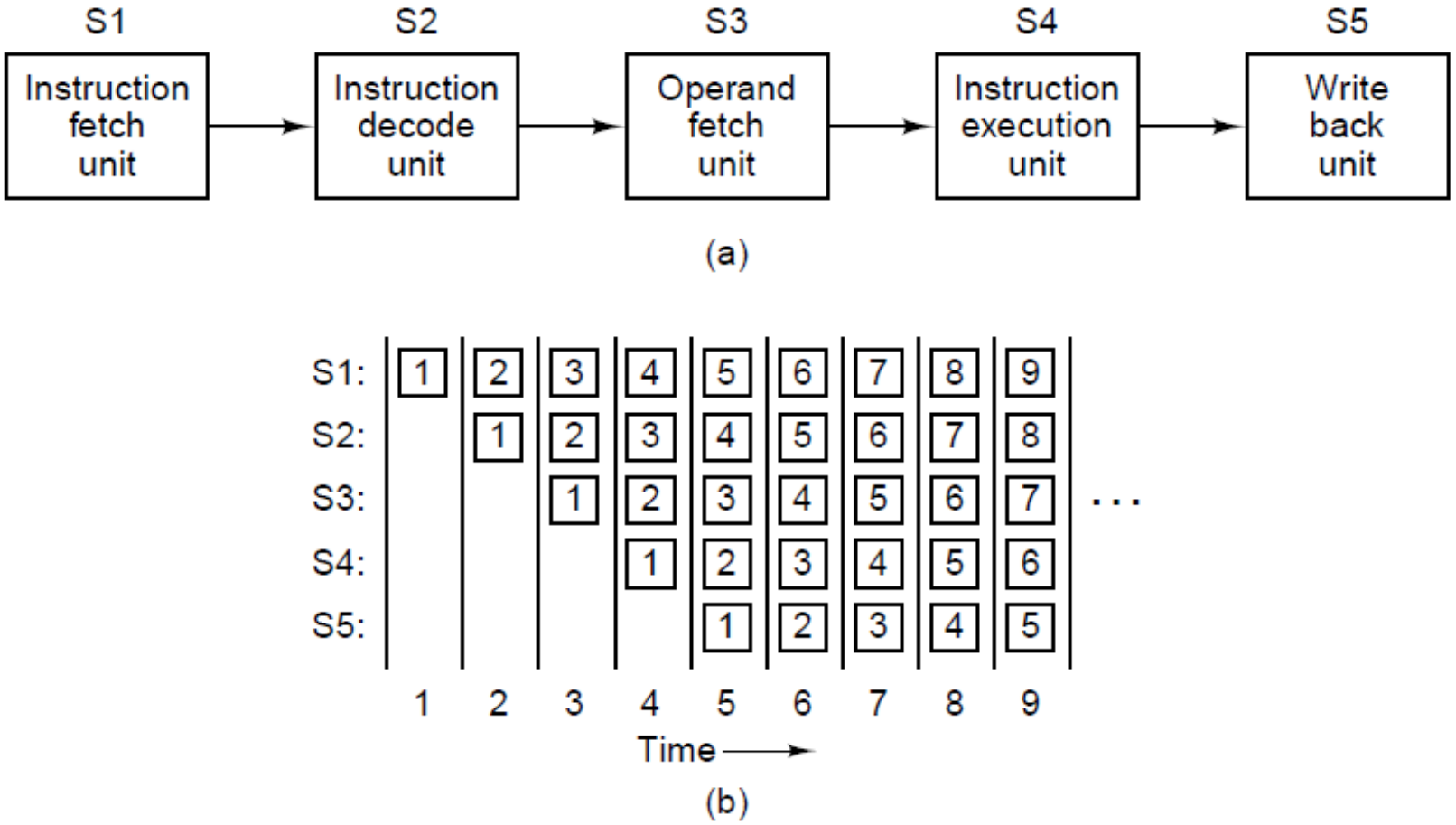
Review: Un-Pipelined Laundry

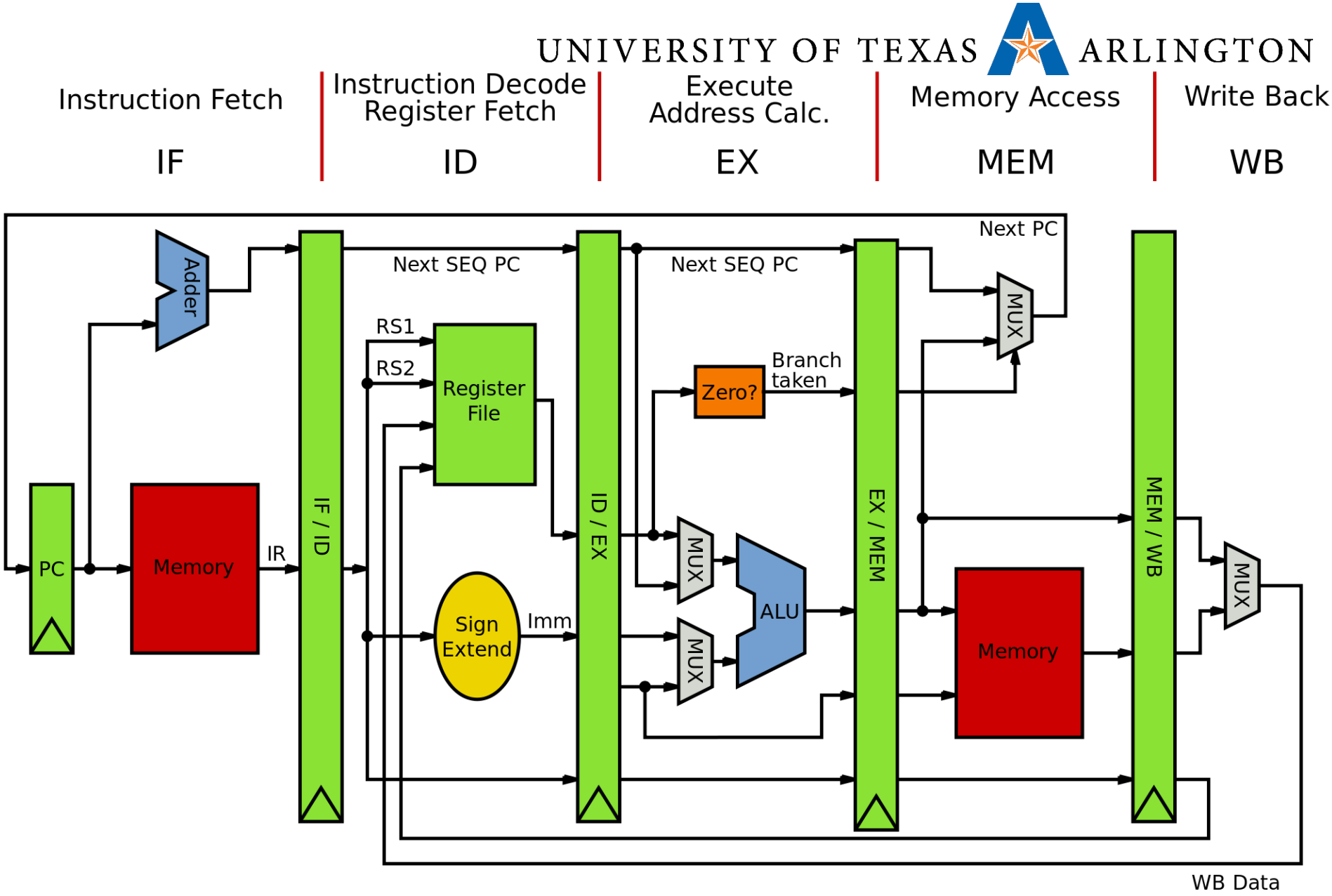


Review: Pipelined Laundry



Review: Pipelining: Instruction-Level Parallelism



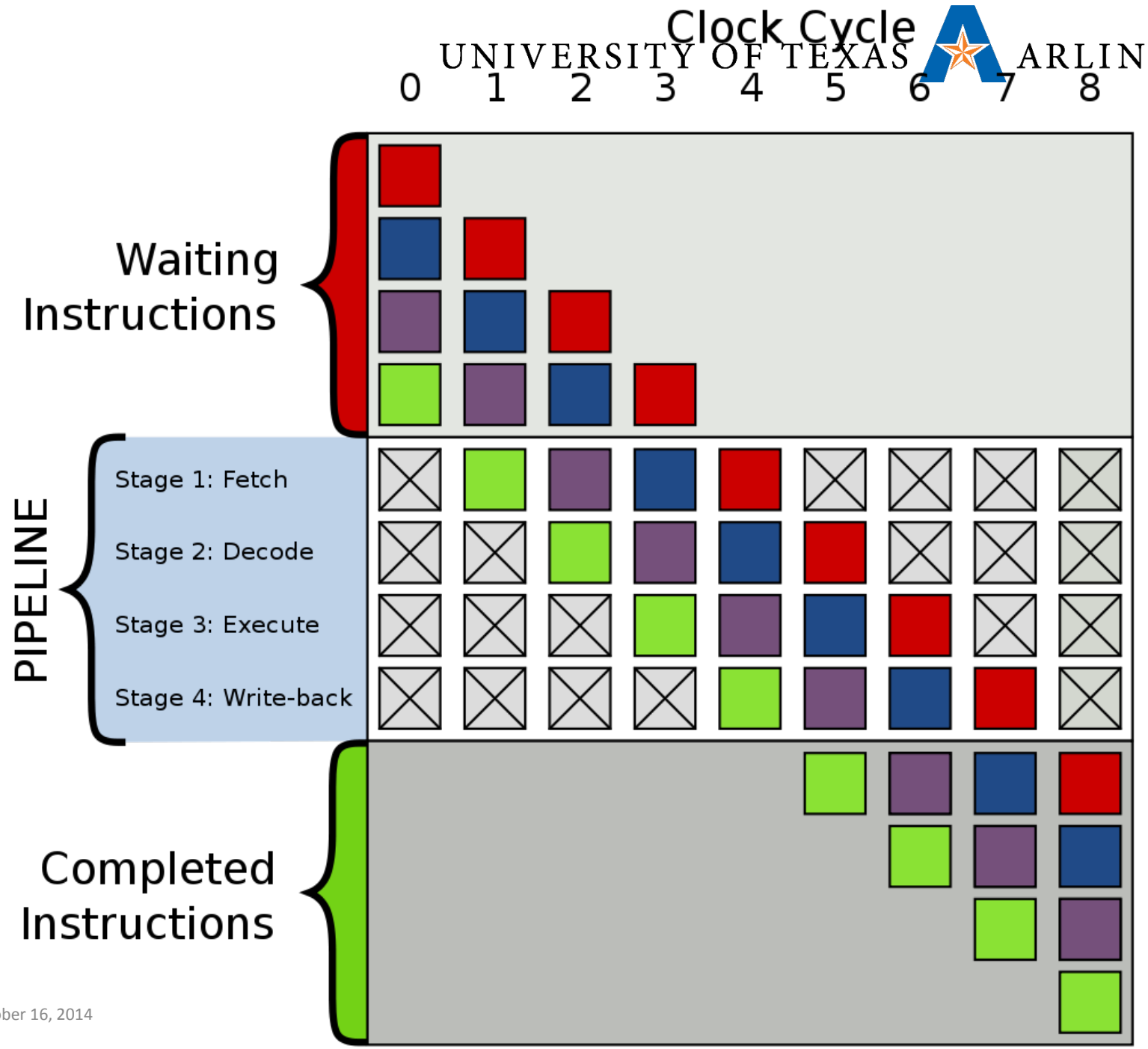

Instruction Fetch
IF
Instruction Decode
Register Fetch
ID
Execute
Address Calc.
EX
Memory Access
MEM
Write Back
WB
Adder
Next SEQ PC
Next SEQ PC
Next PC
MUX
RS1
RS2
Register
File
Branch
taken
Zero?
PC
Memory
IR
IF / ID
ID / EX
EX / MEM
MEM / WB
MUX
MUX
ALU
Sign
Extend
Imm
Memory
MUX
WB Data

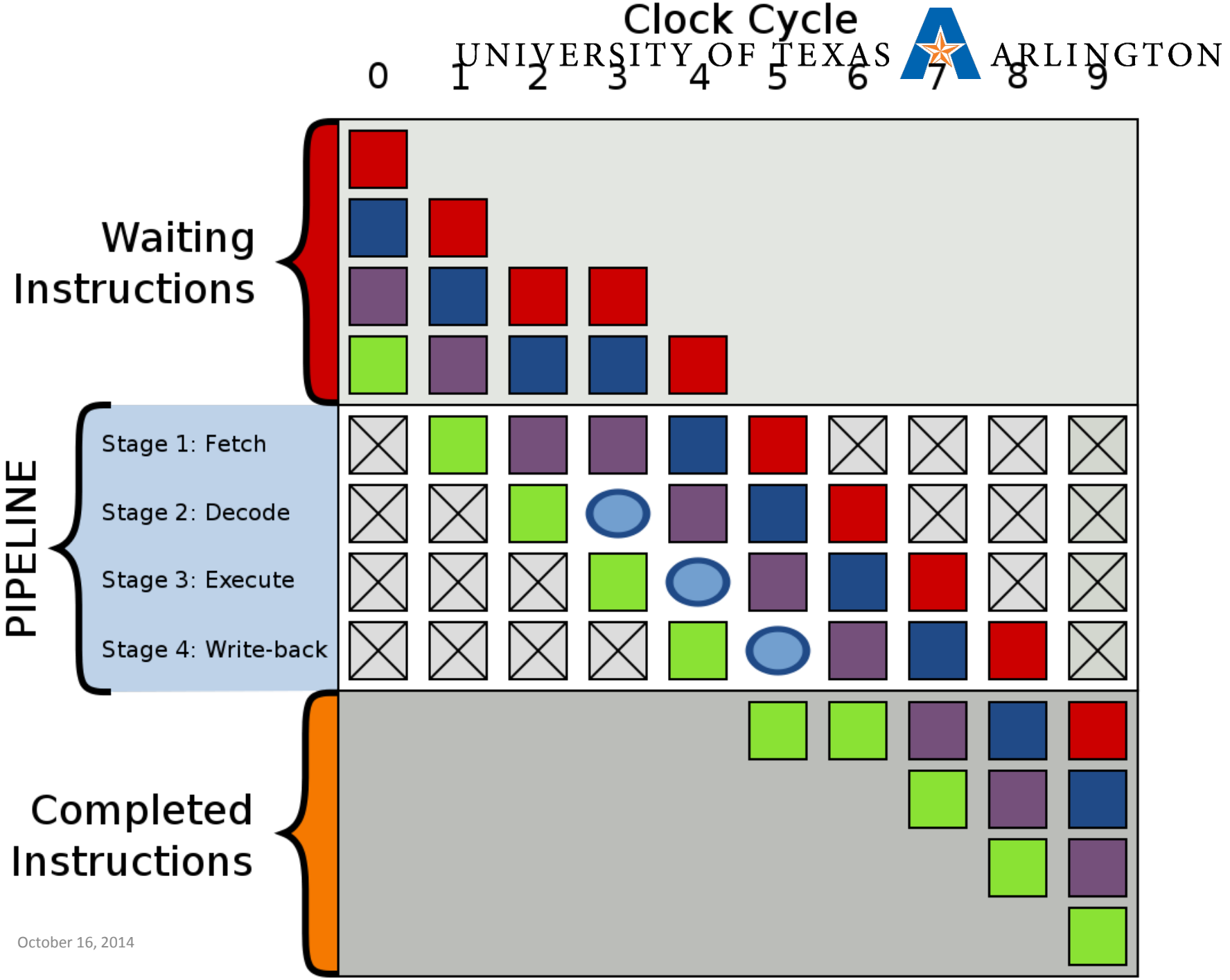
Review: Hazards

- Hazard: next instruction cannot make progress in next cycle
- Data hazards: instruction depends on result of prior instruction completing
  - Example:

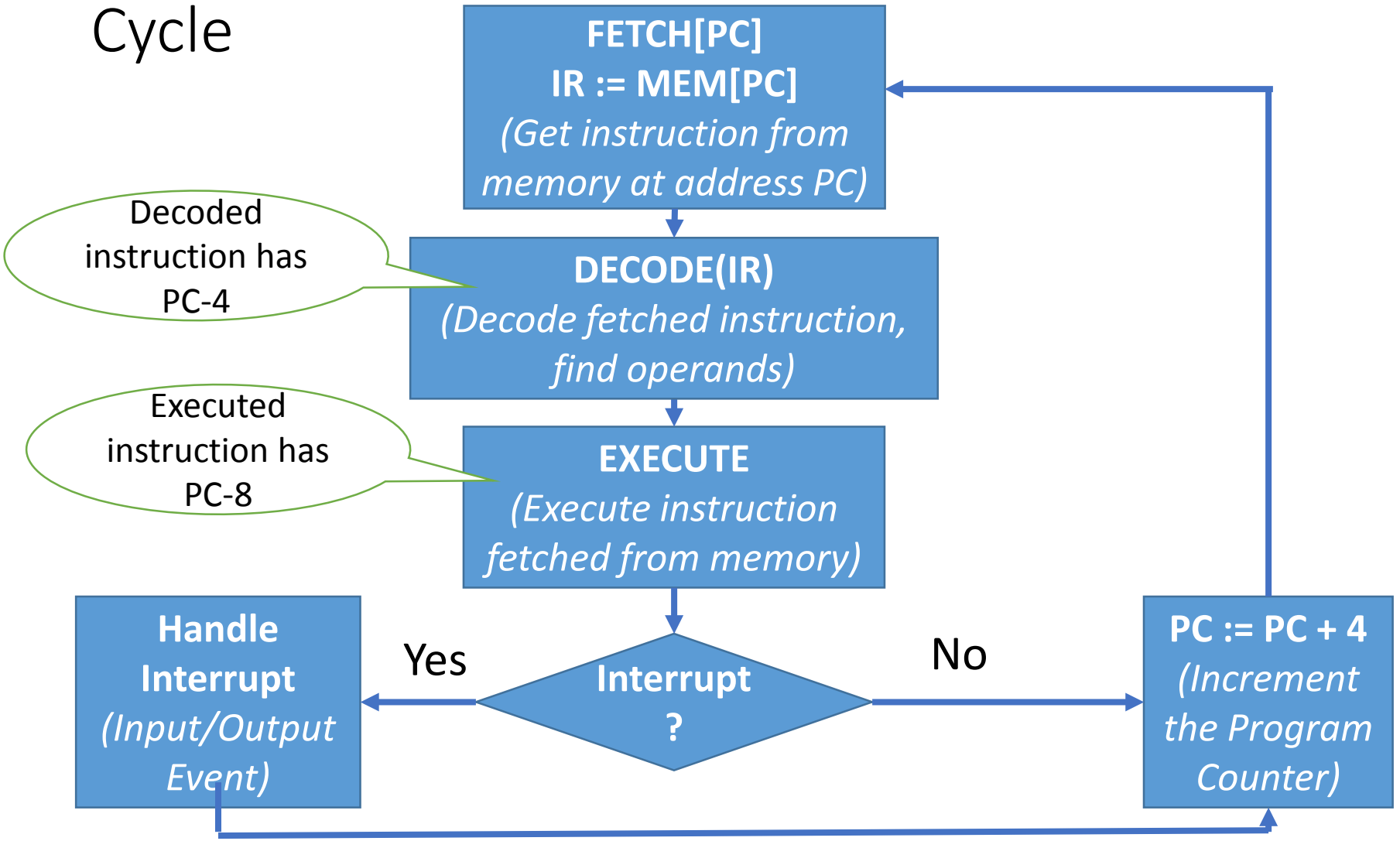
```
ldr r1, =0x1000
str r0, [r1]
ldr r2, [r1]
```

    Problem: `ldr` cannot occur until `str` has completed
  - Solution: stall, out-of-order execution, register forwarding
- Structural Hazards: more than 1 instruction needs to use same processor component (e.g., ALU)
  - Solution: stall, out-of-order execution, …
- Control Hazards: direction of control flow (e.g., branch) depends on prior instructions
  - Solution: stall, branch prediction

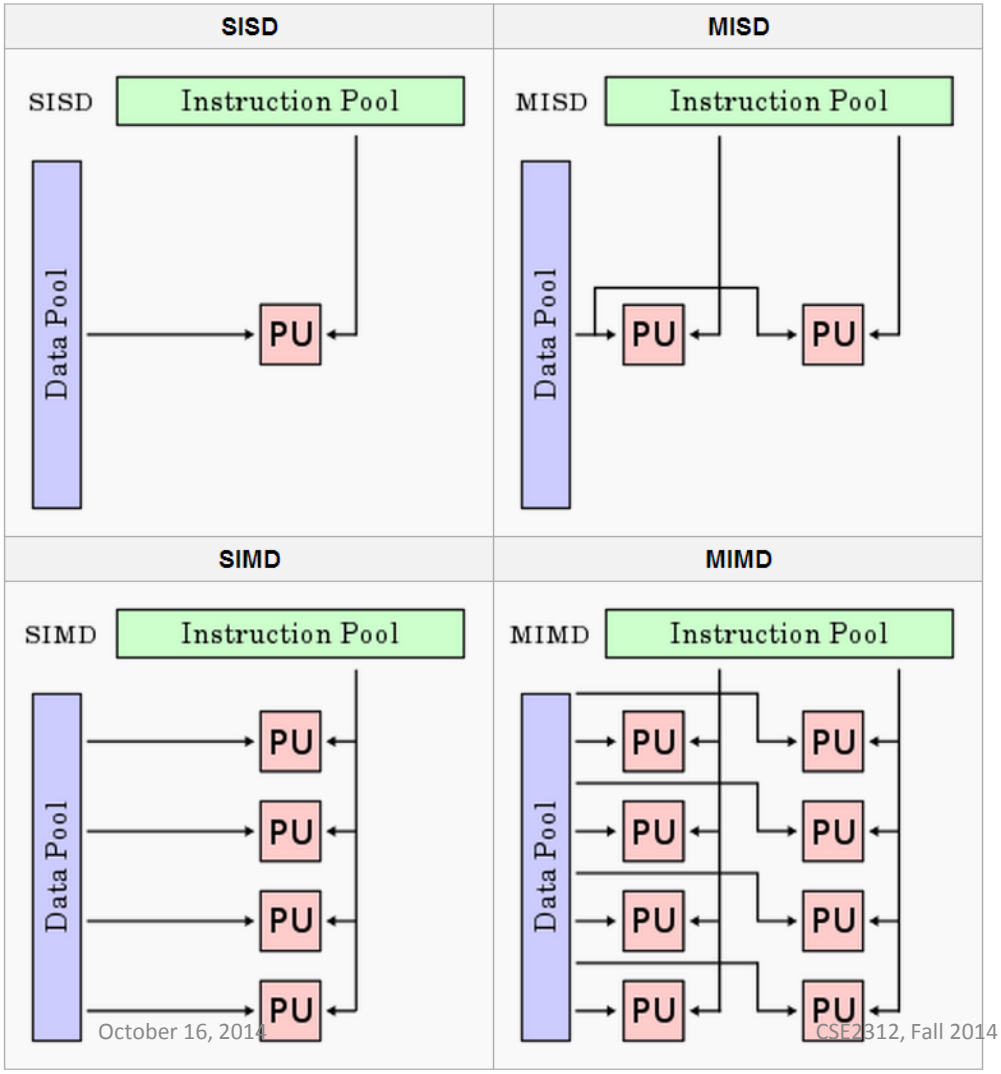

Clock Cycle
0 1 2 3 4 5 6 7 8
Waiting Instructions
PIPELINE
Stage 1: Fetch
Stage 2: Decode
Stage 3: Execute
Stage 4: Write-back
Completed Instructions


Clock Cycle
0
1
2
3
4
5
6
7
8
9
Waiting Instructions
PIPELINE
Stage 1: Fetch
Stage 2: Decode
Stage 3: Execute
Stage 4: Write-back
Completed Instructions

ARM 3-Stage Pipeline Processor Execution Cycle



Review: Flynn's Taxonomy



- SISD: Single Instruction, Single Data
  - Classical Von Neumann
- SIMD: Single Instruction, Multiple Data
  - GPUs
- MISD: Multiple Instruction, Single Data
  - More exotic: fault-tolerant computers using task replication (Space Shuttle flight control computers)
- MIMD: Multiple Instruction, Multiple Data
  - Multiprocessors, multicomputers, server farms, clusters, …

Review: Design Principles for Modern Computers

- All (or most) instructions directly executed by hardware
- Maximize rate at which instructions are issued
  - Issuing instructions rapidly and processing them in parallel leads to higher efficiency, and can compensate up to a degree for longer completion times for each instruction
- Instructions should be easy to decode, especially to figure out what resources they need (so as to issue instructions for which resources are available)

Design Principles for Modern Computers

- Only loads and stores should reference memory
  - Memory is slow to access
  - Loads and stores can be overlapped with other instructions that use the registers and the ALU
- Provide plenty of registers
  - Again, memory is slow
  - So it is good to have enough registers
  - Data can be kept until it is no longer needed (instead of swapping data repeatedly in and out of registers because of limited register space)

Optimizing Fetch-Decode-Execute

1. Fetch next instruction from memory
2. Change program counter to point to next instruction
3. Determine type of instruction just fetched
4. If instruction uses a word in memory, locate it
5. Fetch word, if needed, into a CPU register.
6. Execute instruction.
7. The cycle is completed. Go to step 1 to begin executing the next instruction.

- Long ago, people noticed that **fetch** takes a long time. How can we make up for that delay?

A Simple Two-Step Pipeline

- While one instruction is executed, fetch the next instruction
- Thus, the execution of the next instruction starts **before** the execution of the current instruction finishes
  - Tick 1: fetch instruction 1
  - Tick 2: decode/execute instruction 1, fetch instruction 2
  - Tick 3: decode/execute instruction 2, fetch instruction 3
  - Tick 4: decode/execute instruction 3, fetch instruction 4
  - …
- What do we gain by pipelining? **Speed**
  - No need to spend any time waiting for fetches

Pipelining in General (1)

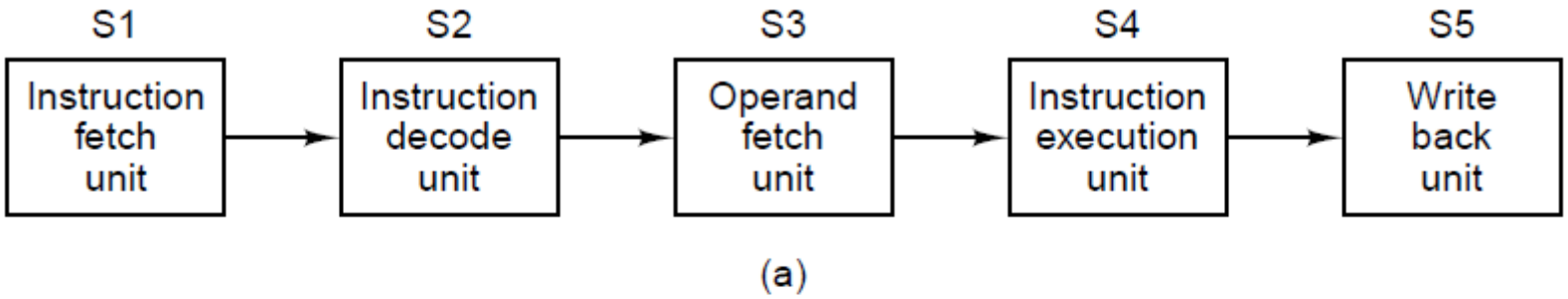
- The instruction execution cycle uses different resources at different steps:
  - Fetch: Memory to fetch the instruction
  - Decode: The decoder to decode the instruction
  - Execute: Move data to ALU input registers
  - Execute: Apply ALU operation to data
  - Execute: Move data from ALU output register

Pipelining in General (2)

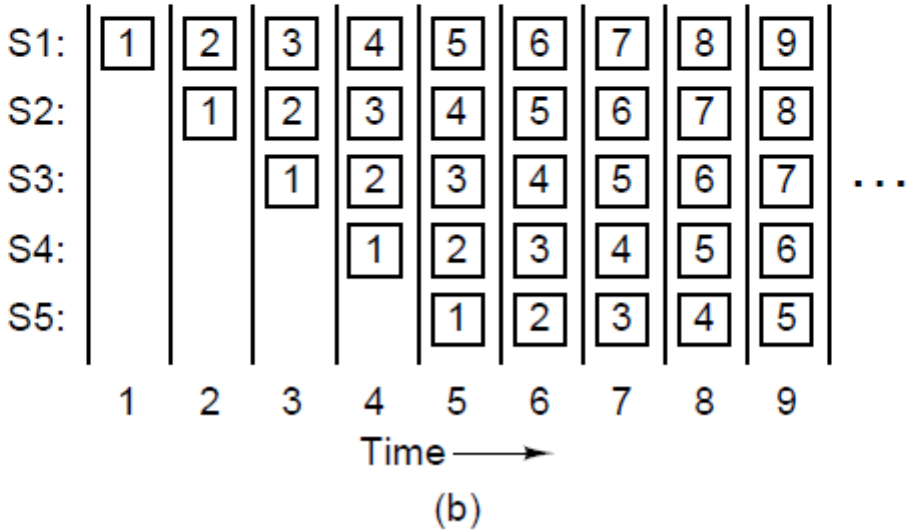
- The instruction execution cycle uses different resources at different steps
- This means that at each execution step, the instruction uses only a small part of the CPU hardware
- The key idea in pipelining is that we can process steps of multiple instructions simultaneously, as long as these steps use different resources
- This idea is also called **instruction-level parallelism** (ILP)
- Modern architectures may use pipelines of 12 or more steps

Illustration of Pipelining

(a) A five-stage pipeline. (b) The state of each stage as a function of time. Nine clock cycles are illustrated



(a)



(b)

Benefits of Pipelining

- Suppose that a single instruction takes 10 nanoseconds to execute, and we have a pipeline of length 5.
  - Without pipelining, we can execute 100 million instructions per second.
  - Using pipelining, in the best case, we can execute ??? instructions per second.
  - Using pipelining, in the worst case, we can execute ??? instructions per second.

Benefits of Pipelining

- Suppose that a single instruction takes 10 nanoseconds to execute, and we have a pipeline of length 5.
  - Without pipelining, we can execute 100 million instructions per second.
  - Using pipelining, in the best case, we can execute 500 million instructions per second.
  - Using pipelining, in the worst case, we can execute 100 instructions per second.
- Are these (best-case number, worst-case number) meaningful measures?

Benefits of Pipelining

- Suppose that a single instruction takes 10 nanoseconds to execute, and we have a pipeline of length 5.
  - Without pipelining, we can execute 100 million instructions per second.
  - Using pipelining, in the best case, we can execute 500 million instructions per second.
  - Using pipelining, in the worst case, we can execute 100 instructions per second.
- Sometimes pipelines cannot be fully utilized.
  - The **average number of instructions per second** is harder to compute (and depends on the code that is used in the simulations), but more useful as a measure.

Issues with Pipelining

- Can you think of a case where pipelines cannot be fully utilized?

Issues with Pipelining: Data Dependencies

- Suppose we have a sequence of instructions, such that each instruction uses the result of the previous one.
- Then, can we evaluate those instructions in a pipeline manner?
- Possibly, but we must wait until the result of the previous instruction is available.
  - Instructions at a certain point in the timeline may pause for a few clock ticks before proceeding.

Example of Data Dependencies

- Let's look at an example of code in assembly.
  - For now we use a made-up assembly language, but it is similar to typical assembly languages we will see later.
- We define three instructions:
  - **add A B C**:
    - Adds contents of registers B and C, stores result in register A.
  - **load A address**:
    - Loads data from the specified memory address to register A.
  - **store A address**:
    - Stores data from register A to the specified memory address.

instruction 1: **load R1 address1**

instruction 2: **load R2 address2**

instruction 3: **add R3 R1 R2**

instruction 4: **add R4 R2 R3**

instruction 5: **store R4 address1**

instruction 6: **load R5 address3**

instruction 7: **load R6 address4**

instruction 8: **add R7 R5 R6**

instruction 9: **add R8 R6 R7**

instruction 10: **store R8 address4**

Step T5: Cannot do operand fetch on instruction 3. The operands of instruction 3 are R1 and R2, and they do not contain the right data until instructions 1 and 2 finish executing (step T6).

Time	T1	T2	T3	T4	**T5**	T6	T7	T8	T9	T10	T11	T12	T13	T14
Instruction fetch	1	2	3	4	**4**	4	5	5	5	6	6	6	7	8
Decode	X	1	2	3	**3**	3	4	4	4	5	5	5	6	7
Operand fetch	X	X	1	2	**X**	X	3	X	X	4	X	X	5	6
Execute operation	X	X	X	1	**2**	X	X	3	X	X	4	X	X	5
Write back result	X	X	X	X	**1**	2	X	X	3	X	X	4	X	X

instruction 1: **load R1 address1**

instruction 2: **load R2 address2**

instruction 3: **add R3 R1 R2**

instruction 4: **add R4 R2 R3**

instruction 5: **store R4 address1**

instruction 6: **load R5 address3**

instruction 7: **load R6 address4**

instruction 8: **add R7 R5 R6**

instruction 9: **add R8 R6 R7**

instruction 10: **store R8 address4**

Step T8: Cannot do operand fetch on instruction 4. One operand of instruction 4 is R3, and it does not contain the right data until instruction3 finishes executing (step T6).

Time	T1	T2	T3	T4	T5	T6	T7	**T8**	T9	T10	T11	T12	T13	T14
Instruction fetch	1	2	3	4	4	4	5	**5**	5	6	6	6	7	8
Decode	X	1	2	3	3	3	4	**4**	4	5	5	5	6	7
Operand fetch	X	X	1	2	X	X	3	**X**	X	4	X	X	5	6
Execute operation	X	X	X	1	2	X	X	**3**	X	X	4	X	X	5
Write back result	X	X	X	X	1	2	X	**X**	3	X	X	4	X	X

instruction 1: **load R1 address1**

instruction 2: **load R2 address2**

instruction 3: **add R3 R1 R2**

instruction 4: **add R4 R2 R3**

instruction 5: **store R4 address1**

instruction 6: **load R5 address3**

instruction 7: **load R6 address4**

instruction 8: **add R7 R5 R6**

instruction 9: **add R8 R6 R7**

instruction 10: **store R8 address4**

Step T11: Cannot do operand fetch on instruction 5. One operand of instruction 5 is R5, which does not contain the right data until instruction 4 finishes executing (step T12).

Time	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	**T11**	T12	T13	T14
Instruction fetch	1	2	3	4	4	4	5	5	5	6	**6**	6	7	8
Decode	X	1	2	3	3	3	4	4	4	5	**5**	5	6	7
Operand fetch	X	X	1	2	X	X	3	X	X	4	**X**	X	5	6
Execute operation	X	X	X	1	2	X	X	3	X	X	**4**	X	X	5
Write back result	X	X	X	X	1	2	X	X	3	X	**X**	4	X	X

instruction 1: **load R1 address1**

instruction 2: **load R2 address2**

instruction 3: **add R3 R1 R2**

instruction 4: **add R4 R2 R3**

instruction 5: **store R4 address1**

instruction 6: **load R5 address3**

instruction 7: **load R6 address4**

instruction 8: **add R7 R5 R6**

instruction 9: **add R8 R6 R7**

instruction 10: **store R8 address4**

Compare to what would happen if we could keep the pipeline always full (which is simply impossible if we execute these instructions in the order in which they are given.

Time	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14
Instruction fetch	1	2	3	4	5	6	7	8	9	10	X	X	X	X
Decode	X	1	2	3	4	5	6	7	8	9	10	X	X	X
Operand fetch	X	X	1	2	3	4	5	6	7	8	9	10	X	X
Execute operation	X	X	X	1	2	3	4	5	6	7	8	9	10	X
Write back result	X	X	X	X	1	2	3	4	5	6	7	8	9	10

instruction 1: **load R1 address1**

instruction 2: **load R2 address2**

instruction 3: **add R3 R1 R2**

instruction 4: **add R4 R2 R3**

instruction 5: **store R4 address1**

instruction 6: **load R5 address3**

instruction 7: **load R6 address4**

instruction 8: **add R7 R5 R6**

instruction 9: **add R8 R6 R7**

instruction 10: **store R8 address4**

Compare to what would happen if we did not use any pipelining whatsoever.

Time	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14
Instruction fetch	1	X	X	X	X	2	X	X	X	X	3	X	X	X
Decode	X	1	X	X	X	X	2	X	X	X	X	3	X	X
Operand fetch	X	X	1	X	X	X	X	2	X	X	X	X	3	X
Execute operation	X	X	X	1	X	X	X	X	2	X	X	X	X	3
Write back result	X	X	X	X	1	X	X	X	X	2	X	X	X	X

instruction 1: **load R1 address1**
instruction 2: **load R2 address2**
instruction 3: **add R3 R1 R2**
instruction 4: **add R4 R2 R3**
instruction 5: **store R4 address1**
instruction 6: **load R5 address3**
instruction 7: **load R6 address4**
instruction 8: **add R7 R5 R6**
instruction 9: **add R8 R6 R7**
instruction 10: **store R8 address4**

There is one trick, widely used, to make pipelining more efficient: **out-of-order execution** of instructions.
The program below is a reordered version of the program above.

instruction 1: **load R1 address1**
instruction 2: **load R2 address2**
instruction 6: **load R5 address3**
instruction 7: **load R6 address4**
instruction 3: **add R3 R1 R2**
instruction 8: **add R7 R5 R6**
instruction 4: **add R4 R2 R3**
instruction 9: **add R8 R6 R7**
instruction 5: **store R4 address1**
instruction 10: **store R8 address4**

Instructions are reordered so that more of them can be executed at the same time. Of course, we must be very careful: **Out-of-order execution should never change the result.**

instruction 1: **load R1 address1**

instruction 2: **load R2 address2**

instruction 6: **load R5 address3**

instruction 7: **load R6 address4**

instruction 3: **add R3 R1 R2**

instruction 8: **add R7 R5 R6**

instruction 4: **add R4 R2 R3**

instruction 9: **add R8 R6 R7**

instruction 5: **store R4 address1**

instruction 10: **store R8 address4**

Execution of reordered instructions: the pipeline gets more fully utilized.

Time	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14
Instruction fetch	1	2	6	7	3	8	4	4	9	5	5	10	10	X
Decode	X	1	2	6	7	3	8	8	4	9	9	5	5	10
Operand fetch	X	X	1	2	6	7	3	X	8	4	X	9	X	5
Execute operation	X	X	X	1	2	6	7	3	X	8	4	X	9	X
Write back result	X	X	X	X	1	2	6	7	3	X	8	4	X	9

Out-of-Order Execution

- For more information, read:

  http://en.wikipedia.org/wiki/Out-of-order_execution

- Key idea:
  - Fetched instructions do not go directly to the pipeline. Instead, they join an **instruction queue**
  - An instruction is held in that queue until its operands are available. Then, it is allowed to enter the pipeline
- Out-of-order execution requires more complicated CPUs
- Now standard in desktop/laptop processors

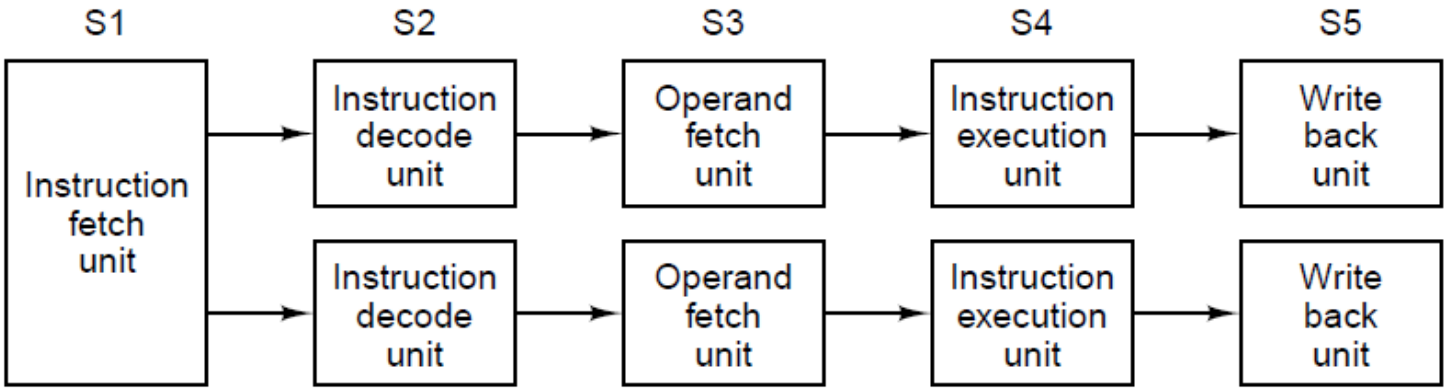
Issues with Pipelining: Branching

- Suppose we have a branching statement (if, else).
- Until that statement is executed, the next statement is not known. Thus, the CPU does not know what to put in the pipeline after the branching statement.
- Common solution: guess (formal term: **branch prediction**).
- If the guess is wrong, **undo the work that was based on guessing**, and resume.
- For more information, read:

  http://en.wikipedia.org/wiki/Branch_predictor

Superscalar Architectures (1)

- Dual five-stage pipelines with common instruction fetch unit
  - Fetches pairs of instructions together and puts each into its own pipeline
  - Two instructions must not conflict over resource usage
  - Neither must depend on the results of others



Superscalar Architectures (2)

- If one pipeline is good, two pipelines are even better.
  - Is that correct?

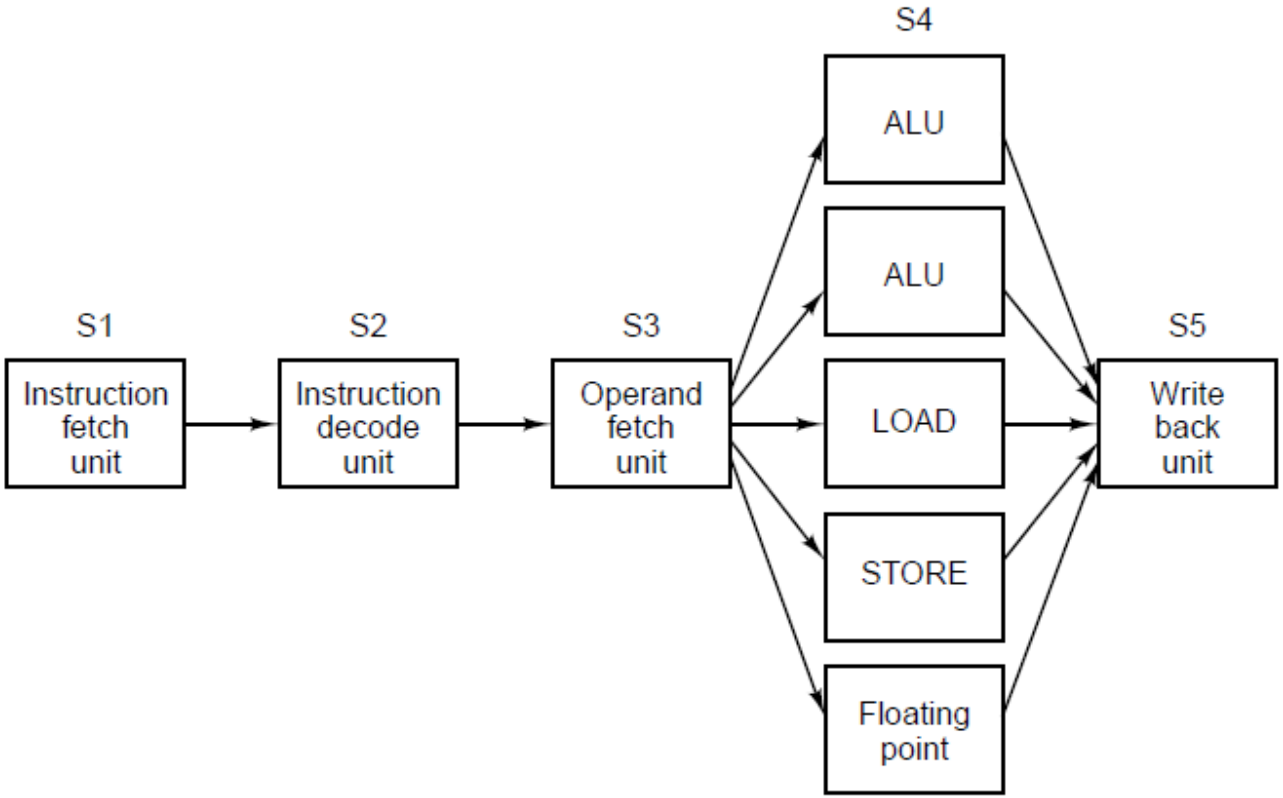
Superscalar Architectures (2)

- If one pipeline is good, two pipelines are even better.
  - Is that correct? Yes, it is fairly easy to prove that moving from one to two pipelines will never hurt performance, and on average it will improve performance.
- Same issues involved when using a single pipeline arise here:
  - Cannot execute instructions until their operands are available.
  - Cannot execute instructions in parallel if they use the same resource (i.e., write result on the same register at the same time).
  - Branch predictions are used (and may go wrong).
  - Out-of-order execution is widely used, improves efficiency.

Superscalar Architectures (3)

Figure 2-6. A superscalar processor with five functional units.

Intuition: S3 stage issues instructions considerably faster than the S4 stage can execute them



Superscalar Architectures (4)

- The previous figure assumes that S3 (operand fetch) works match faster than S4 (execution).
  - That is indeed the typical case.
- This type of architecture requires the CPU to have multiple units for the same function.
  - For example, multiple ALUs.
- This type of architecture is nowadays common, due to improved hardware capabilities.

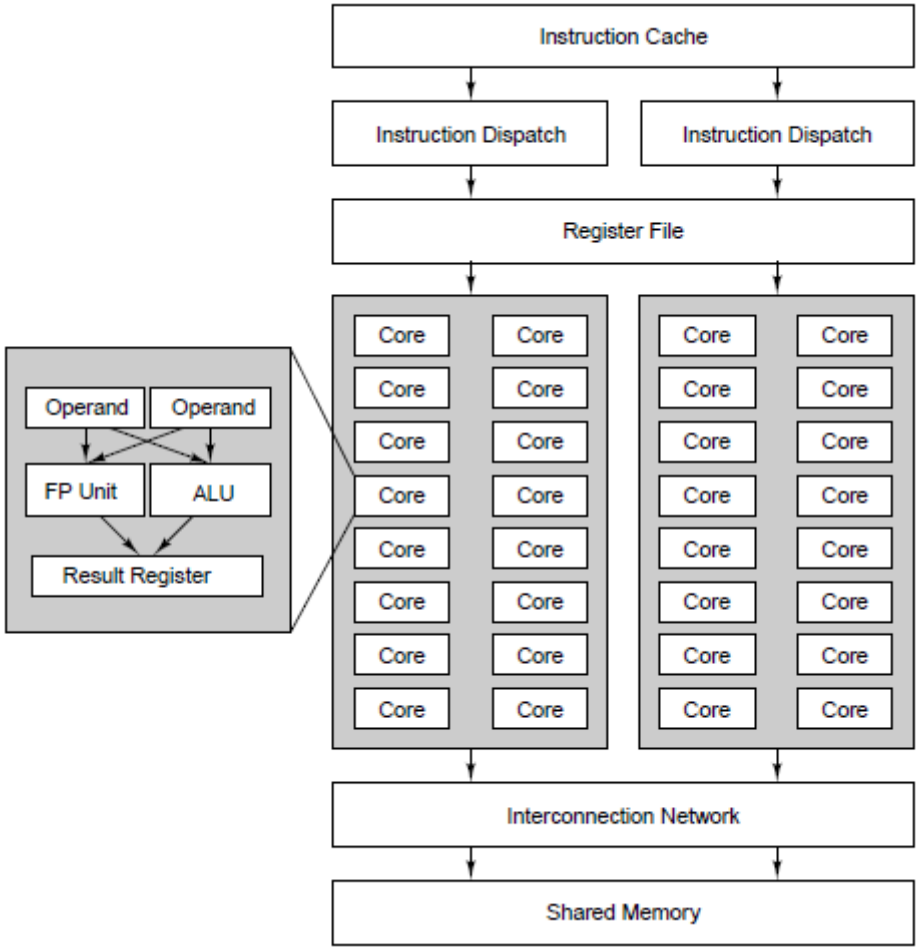
Processor-Level Parallelism

- The idea behind Processor-Level Parallelism:
  - multiple processors are better than a single processor.
- However, there are several intermediate designs between a single processor and multiple processors:
  - Data parallel computers.
    - Single Instruction-stream Multiple Data-stream (SIMD) processors.
    - Vector Processors.
  - Multiprocessors.
  - Multiple computers.

Data Parallelism

- Many problems, especially in the physical sciences, engineering, and graphics, involve performing the same exact calculations on different data.
- Example: making an image brighter.
  - An image is a 2-dimensional array of pixels.
  - Each pixel contains three numbers: R, G, B, describing the color of that pixel (how much red, green, and blue it contains).
  - We perform the same numerical operation (adding a constant) on thousands (or millions) of different pixels.
- Graphics cards and video game platforms perform such operations on a regular basis.

Data Parallel Computers



SIMD Processors

- SIMD stands for **Single Instruction-stream Multiple Data-stream**.
- There are multiple processors.
- There is a single control unit, executing a single sequence of instructions.
- Each instruction in the sequence is broadcast to all processors.
- Each processor applies the instruction on its own local data, from its own memory.
- Why is this any better than just using multiple processors?

SIMD Processors

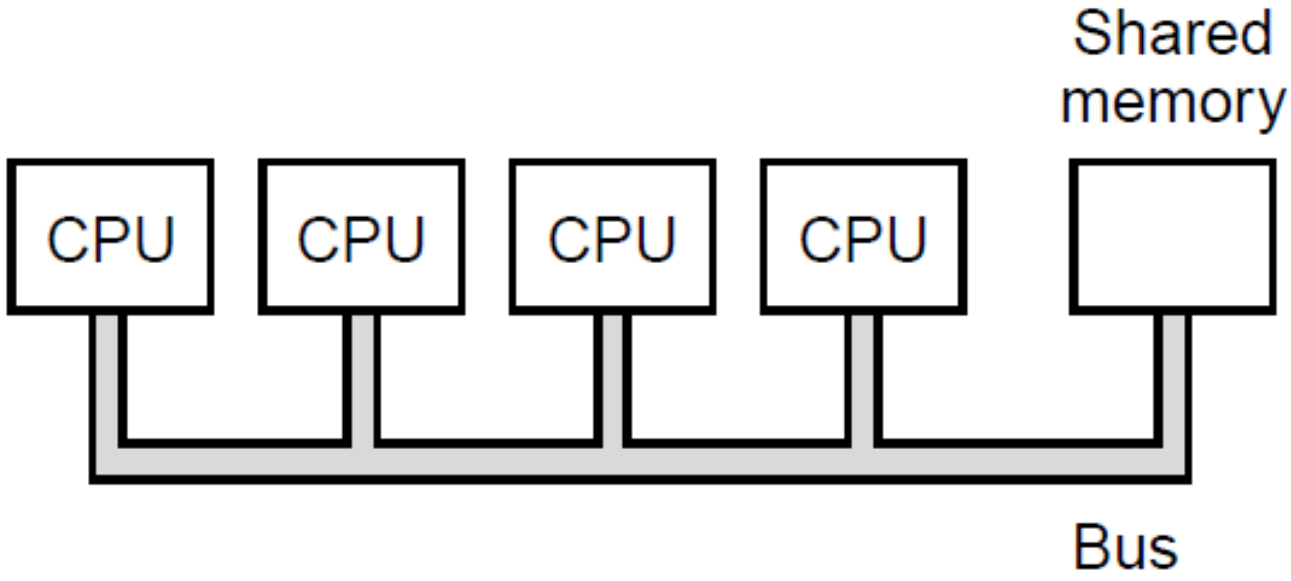
- The multiple processors in a SIMD architecture are greatly simplified, because they do not need a control unit.
- For example, it is a lot cheaper to design and mass-produce a SIMD machine with 1000 processors, than a regular machine with 1000 processors, since the SIMD processors do not need control units.

Vector Processors

- Applicable in exactly the same case as SIMD processors:
  - when the same sequence of instructions is applied to many sets of different data.
- This problem allows for very large pipelines.
  - applying the same set of instructions on different data means there are no dependencies/conflicts among instructions.
  - To support these very large pipelines, the CPU needs a large number of registers, to hold the data of all instructions in the pipeline.

Multiprocessors (1)

- Figure 2-8. (a) A single-bus multiprocessor.

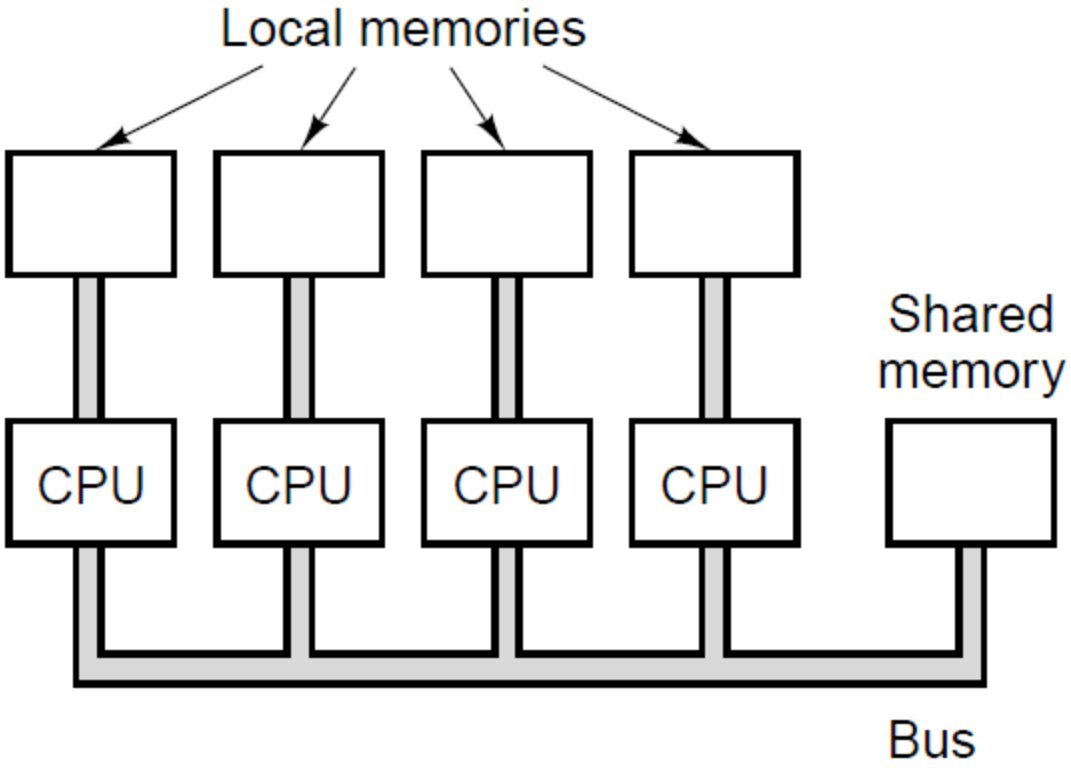


Multiprocessors (2)

- The design in the previous slide uses a single bus, connecting multiple CPUs with the same memory.
- Advantage: compared to SIMD machines and vector processors, multiprocessor machines can execute different instructions at the same time.
- Advantage: having a single memory makes programming easy (compared to multiple memories).
- Disadvantage: if multiple CPUs try to access the main memory at the same time, the bus cannot support all of them simultaneously.
  - Some CPUs have to wait, so they do not get used at 100% efficiency.

Multicomputers (1)

- Figure 2-8(b) A multicomputer with local memories.



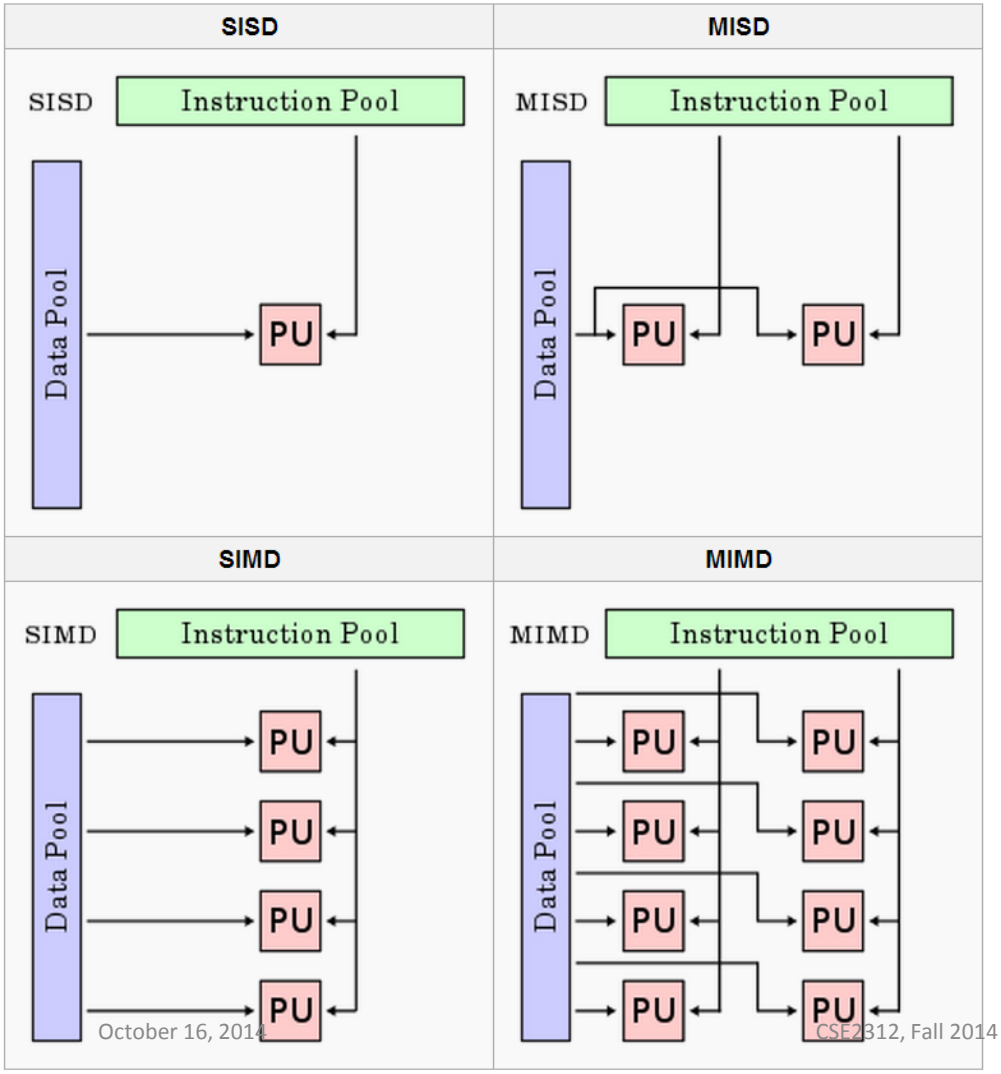
Multicomputers (2)

- At some point (close to 256 processors these days), multiprocessors reach their limit.
  - Hard to connect that many processors to a single memory.
  - Hard to avoid conflicts (multiple CPUs reading/writing the same memory location).
- Multicomputers are the logical next step:
  - Multiple processors.
  - Each processor has its own bus and memory.
- Easy to scale, for example to 250,000 computers.

Multicomputers (3)

- Major disadvantage of microcomputers: they are difficult to program:
  - a single C or Java program is not sufficient.
  - hard to divide instructions and data appropriately.
  - hard to combine multiple results together.

Flynn's Taxonomy



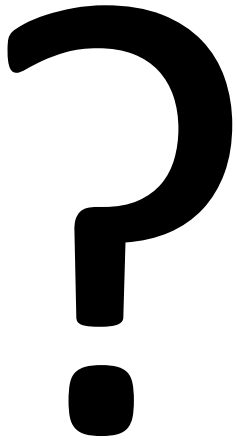
- SISD: Single Instruction, Single Data
  - Classical Von Neumann
- SIMD: Single Instruction, Multiple Data
  - GPUs
- MISD: Multiple Instruction, Single Data
  - More exotic: fault-tolerant computers using task replication (Space Shuttle flight control computers)
- MIMD: Multiple Instruction, Multiple Data
  - Multiprocessors, multicomputers, server farms, clusters, …

Summary

- Pipelines
  - Instruction-level parallelism
    - Running pieces of several instructions simultaneously to make the most use of available fixed resources (think laundry)
  - Other forms of parallelism: Flynn's taxonomy

- Know what make does
- Know how to start QEMU
- Know how to start GDB
- Start learning how to interact and debug with GDB
  - Saw example of debugging the stack, etc.

Questions?



UNIVERSITY OF TEXAS ARLINGTON

More on Pipelining

CSE 2312
Computer Organization and Assembly Language Programming
Vassilis Athitsos
University of Texas at Arlington

Fetch-Decode-Execute Cycle in Detail

1. Fetch next instruction from memory
2. Change program counter to point to next instruction
3. Determine type of instruction just fetched
4. If instruction uses a word in memory, locate it
5. Fetch word, if needed, into a CPU register.
6. Execute instruction.
7. The clock cycle is completed. Go to step 1 to begin executing the next instruction.

Toy ISA Instructions

- **add A B C**:
  - Adds contents of registers B and C, stores result in register A.
- **addi A C N**:
  - Adds integer N to contents of register C, stores result in register A.
- **load A address**:
  - Loads data from the specified memory address to register A.
- **store A address**:
  - Stores data from register A to the specified memory address.
- **goto line**:
  - Set the instruction counter to the specified line. That line should be executed next.
- **if A line**:
  - If the contents of register A are NOT 0, set the instruction counter to the specified line. That line should be be executed next.

Defining Pipeline Behavior

- In the following slides, we will explicitly define how each instruction goes through the pipeline.
- This is a toy ISA that we have just made up, so the following conventions are designed to be simple, and easy to apply.
- You may find that, in some cases, we could have followed other conventions that would make execution even more efficient.

Pipeline Steps for: **add A B C**

- **Fetch Step:**
- **Decode Step:**
- **Operand Fetch Step:**
- **Execution Step:**
- **Output Save Step:**

- **NOTES:**

Pipeline Steps for: **add A B C**

- **Fetch Step:** Fetch instruction from memory location specified by PC. Increment PC to point to the next instruction.
- **Decode Step:** Determine that this statement uses the ALU, takes input from registers B and C, and modifies register A.
- **Operand Fetch Step:** Copy contents of registers B and C to ALU input registers.
- **Execution Step:** The ALU unit performs addition.
- **Output Save Step:** The result of the addition is copied to register A.

- **NOTES:** This instruction must wait at the decode step until all previous instructions have finished modifying the contents of registers B and C.

Pipeline Steps for: addi A C N

- **Fetch Step:**
- **Decode Step:**
- **Operand Fetch Step:**
- **Execution Step:**
- **Output Save Step:**

- **NOTES:**

Pipeline Steps for: **addi A C N**

- **Fetch Step:** Fetch instruction from memory location specified by PC. Increment PC to point to the next instruction.
- **Decode Step:** Determine that this statement uses the ALU, takes input from register C, and modifies register A.
- **Operand Fetch Step:** Copy content of register C into one ALU input register, copy integer N into the other ALU input register.
- **Execution Step:** The ALU unit performs addition.
- **Output Save Step:** The result of the addition is copied to register A.

- **NOTES:** This instruction must wait at the decode step until all previous instructions have finished modifying the contents of register C.

Pipeline Steps for: load A address

- **Fetch Step:**
- **Decode Step:**
- **Operand Fetch Step:**
- **Execution Step:**
- **Output Save Step:**

- **NOTES:**

Pipeline Steps for: load A address

- **Fetch Step:** Fetch instruction from memory location specified by PC. Increment PC to point to the next instruction.
- **Decode Step:** Determine that this statement accesses memory, takes input from **address**, and modifies register A.
- **Operand Fetch Step:** Not applicable for this instruction.
- **Execution Step:** The bus brings to the CPU the contents of **address**.
- **Output Save Step:** The data brought by the bus is copied to register A.

- **NOTES:** This instruction must wait at the decode step until all previous instructions have finished modifying the contents of **address**.

Pipeline Steps for: **store A address**

- **Fetch Step:**
- **Decode Step:**
- **Operand Fetch Step:**
- **Execution Step:**
- **Output Save Step:**

- **NOTES:**

Pipeline Steps for: **store A address**

- **Fetch Step:** Fetch instruction from memory location specified by PC. Increment PC to point to the next instruction.
- **Decode Step:** Determine that this statement accesses memory, takes input from register A, and modifies **address**.
- **Operand Fetch Step:** Not applicable for this instruction.
- **Execution Step:** The bus receives the contents of register A from the CPU.
- **Output Save Step:** The bus saves the data at **address**.

- **NOTES:** This instruction must wait at the decode step until all previous instructions have finished modifying the contents of register A.

Pipeline Steps for: goto line

- **Fetch Step:**
- **Decode Step:**
- **Operand Fetch Step:**
- **Execution Step:**
- **Output Save Step:**

- **NOTES:**

Pipeline Steps for: **goto line**

- **Fetch Step:** Fetch instruction from memory location specified by PC. Increment PC to point to the next instruction.
- **Decode Step:** Determine that this statement is a **goto**. Flush (erase) what is stored at the fetch step in the pipeline.
- **Operand Fetch Step:** Not applicable for this instruction.
- **Execution Step:** Not applicable for this instruction.
- **Output Save Step:** The program counter (PC) is set to the specified **line**.

- **NOTES:** See next slide.

Pipeline Steps for: **goto line**

- **NOTES:** When a **goto** instruction completes the decode step:
  - The pipeline **stops receiving** any new instructions. However, instructions that entered the pipeline **before** the **goto** instruction continue normal execution.
  - The pipeline ignores and does not process any further the instruction that was fetched while the **goto** instruction was decoded.
- Fetching statements resumes as soon as the **goto** instruction has finished executing, i.e., when the **goto** instruction **has completed** the output save step.

Pipeline Steps for: if A line

- **Fetch Step:**
- **Decode Step:**
- **Operand Fetch Step:**
- **Execution Step:**
- **Output Save Step:**

- **NOTES:**

Pipeline Steps for: **if A line**

- **Fetch Step:** Fetch instruction from memory location specified by PC. Increment PC to point to the next instruction.
- **Decode Step:** Determine that this statement is an **if** and that it accesses register **A**. Flush (erase) what is stored at the fetch step in the pipeline.
- **Operand Fetch Step:** Copy contents of register A to first ALU input register.
- **Execution Step:** The ALU compares the first input register with 0, and outputs 0 if the input register equals 0, outputs 1 otherwise.
- **Output Save Step:** If the ALU output is 1, the program counter (PC) is set to the specified **line**. Nothing done otherwise.

- **NOTES:** See next slide.

Pipeline Steps for: **if A line**

- **NOTE 1:** an **if** instruction must wait at the decode step until all previous instructions have finished modifying register A.
- When an **if** instruction completes the decode step:
  - The pipeline **stops receiving** any new instructions. However, instructions that entered the pipeline **before** the **if** instruction continue normal execution.
  - The pipeline erases and does not process any further the instruction that was fetched while the **if** instruction was decoded.
- Fetching statements resumes as soon as the **if** instruction has finished executing, i.e., when the **if** instruction **has completed** the output save step.

Pipeline Execution: An Example

- Consider the program on the right.
- The previous specifications define how this program is executed step-by-step through the pipeline.
- To trace the execution, we need to specify the inputs to the program.
- Program inputs:

- Program outputs:

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

Pipeline Execution: An Example

- Consider the program on the right.
- The previous specifications define how this program is executed step-by-step through the pipeline.
- To trace the execution, we need to specify the inputs to the program.
- Program inputs:
  - address1, let's assume it contains 0.
  - address2, let's assume it contains 10.
- Program outputs:
  - address10
  - address11
  - address12

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

line 1: **load R2 address2** line 2: **load R1 address1** line 3: **if R1 6** line 4: **addi R3 R1 20** line 5: **goto 7**	line 6: **addi R3 R1 10** line 7: **addi R4 R2 5** line 8: **store R4 address10** line 9: **addi R5 R2 30** line 10: **store R5 address11**	line 11: **add R8 R2 R3** line 12: **store R8 address12**

Time	Fetch	Decode	Operand Fetch	ALU exec.	Output Save	PC	Notes

line 1: **load R2 address2** line 2: **load R1 address1** line 3: **if R1 6** line 4: **addi R3 R1 20** line 5: **goto 7**	line 6: **addi R3 R1 10** line 7: **addi R4 R2 5** line 8: **store R4 address10** line 9: **addi R5 R2 30** line 10: **store R5 address11**	line 11: **add R8 R2 R3** line 12: **store R8 address12**

Time	Fetch	Decode	Operand Fetch	ALU exec.	Output Save	PC	Notes
1	1	X	X	X	X	1	

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

Time	Fetch	Decode	Operand Fetch	ALU exec.	Output Save	PC	Notes
1	1	X	X	X	X	1	
2	2	1	X	X	X	2	

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

Time	Fetch	Decode	Operand Fetch	ALU exec.	Output Save	PC	Notes
1	1	X	X	X	X	1	
2	2	1	X	X	X	2	
3	3	2	1	X	X	3	

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

Time	Fetch	Decode	Operand Fetch	ALU exec.	Output Save	PC	Notes
1	1	X	X	X	X	1	
2	2	1	X	X	X	2	
3	3	2	1	X	X	3	
4	4	3	2	1	X	4	

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

Time	Fetch	Decode	Operand Fetch	ALU exec.	Output Save	PC	Notes
1	1	X	X	X	X	1	
2	2	1	X	X	X	2	
3	3	2	1	X	X	3	
4	4	3	2	1	X	4	
5	4	3	X	2	1	4	line 3 waits for line 2 to finish.

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

Time	Fetch	Decode	Operand Fetch	ALU exec.	Output Save	PC	Notes
1	1	X	X	X	X	1	
2	2	1	X	X	X	2	
3	3	2	1	X	X	3	
4	4	3	2	1	X	4	
5	4	3	X	2	1	4	line 3 waits for line 2 to finish.
6	4	3	X	X	2	4	
7							
8							
9							

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

Time	Fetch	Decode	Operand Fetch	ALU exec.	Output Save	PC	Notes
1	1	X	X	X	X	1	
2	2	1	X	X	X	2	
3	3	2	1	X	X	3	
4	4	3	2	1	X	4	
5	4	3	X	2	1	4	line 3 waits for line 2 to finish.
6	4	3	X	X	2	4	
7	X	X	3	X	X	4	line 3 moves on. **if** detected. Stop fetching, flush line 4 from fetch step.
8							
9							

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

Time	Fetch	Decode	Operand Fetch	ALU exec.	Output Save	PC	Notes
1	1	X	X	X	X	1	
2	2	1	X	X	X	2	
3	3	2	1	X	X	3	
4	4	3	2	1	X	4	
5	4	3	X	2	1	4	line 3 waits for line 2 to finish.
6	4	3	X	X	2	4	
7	X	X	3	X	X	4	line 3 moves on. **if** detected. Stop fetching, flush line 4 from fetch step.
8	X	X	X	3	X	4	
9							

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

Time	Fetch	Decode	Operand Fetch	ALU exec.	Output Save	PC	Notes
1	1	X	X	X	X	1	
2	2	1	X	X	X	2	
3	3	2	1	X	X	3	
4	4	3	2	1	X	4	
5	4	3	X	2	1	4	line 3 waits for line 2 to finish.
6	4	3	X	X	2	4	
7	X	X	3	X	X	4	line 3 moves on. **if** detected. Stop fetching, flush line 4 from fetch step.
8	X	X	X	3	X	4	
9	X	X	X	X	3	4	

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

Time	Fetch	Decode	Operand Fetch	ALU exec.	Output Save	PC	Notes
9	X	X	X	X	3	4	
10	4	X	X	X	X	4	**if** has finished, PC does **NOT** change.
11	5	4	X	X	X	5	
12	6	5	4	X	X	6	
13	X	X	5	4	X	X	**goto** detected. Stop fetching, flush line 6 from fetch step.
14	X	X	X	5	4	X	
15	X	X	X	X	5	X	
16	7	X	X	X	X	7	**goto** has finished, PC set to 7.
17	8	7	X	X	X	8	

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

Time	Fetch	Decode	Operand Fetch	ALU exec.	Output Save	PC	Notes
17	8	7	X	X	X	8	
18	9	8	7	X	X	9	
19	9	8	X	7	X	9	line 8 waits for line 7 to finish.
20	9	8	X	X	7	9	
21	10	9	8	X	X	10	line 8 moves on.
22	11	10	9	8	X	11	
23	11	10	X	9	8	11	line 10 waits for line 9 to finish.
24	11	10	X	X	9	11	
25	12	11	10	X	X	12	line 10 moves on.

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

Time	Fetch	Decode	Operand Fetch	ALU exec.	Output Save	PC	Notes
25	12	11	10	X	X	12	line 10 moves on.
26	X	12	11	10	X	X	no more instructions to fetch.
27	X	12	X	11	X	X	line 12 waits for line 11 to finish.
28	X	12	X	X	11	X	
29	X	X	12	X	X	X	line 12 moves on.
30	X	X	X	12	X	X	
31	X	X	X	X	12	X	
32							program execution has finished!

Reordering Instructions

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

- Reordering of instructions can be done by a compiler, as long as the compiler knows how instructions are executed.
- The goal of reordering is to obtain more efficient execution through the pipeline, by reducing dependencies.
- Obviously, reordering is not allowed to change the **meaning** of the program.
- What is the **meaning** of a program?

Meaning of a Program

```
line 1: load R2 address2
line 2: load R1 address1
line 3: if R1 6
line 4: addi R3 R1 20
line 5: goto 7
line 6: addi R3 R1 10
line 7: addi R4 R2 5
line 8: store R4 address10
line 9: addi R5 R2 30
line 10: store R5 address11
line 11: add R8 R2 R3
line 12: store R8 address12
```

- What is the **meaning** of a program?
- A program can be modeled mathematically as a function, that takes specific input and produces specific output.
- In this program, what is the input? Where is information stored that the program accesses?
- What is the output? What is information left behind by the program?

Meaning of a Program

```
line 1: load R2 address2
line 2: load R1 address1
line 3: if R1 6
line 4: addi R3 R1 20
line 5: goto 7
line 6: addi R3 R1 10
line 7: addi R4 R2 5
line 8: store R4 address10
line 9: addi R5 R2 30
line 10: store R5 address11
line 11: add R8 R2 R3
line 12: store R8 address12
```

- What is the **meaning** of a program?
- A program can be modeled mathematically as a function, that takes specific input and produces specific output.
- In this program, what is the input? Where is information stored that the program accesses?
  - address1 and address2.
- What is the output? What is information left behind by the program?
  - address10, address11, address12.

Reordering Instructions

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

- Reordering is not allowed to change the **meaning** of a program.
- Therefore, when given the same input as the original program, the re-ordered program must produce same output as the original program.
- Therefore, the re-ordered program must ALWAYS leave the same results as the original program on address10, address11, address12, as long as it starts with the same contents as the original program on address1 and address2.

Reordering Instructions

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

- Reordering of instructions can be done by a compiler, as long as the compiler knows how instructions are executed.
- How can we rearrange the order of instructions?
- Heuristic approach: when we find an instruction A that needs to wait on instruction B:
  - See if instruction B can be moved earlier.
  - See if some later instructions can be moved ahead of instruction A.

Reordering Instructions

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

- What is the first instruction that has to wait?
- What can we do for that case?

Reordering Instructions

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

- What is the first instruction that has to wait?
  - line 3 needs to wait on line 2.
- What can we do for that case?
  - Swap line 2 and line 1, so that line 2 happens earlier.

Reordering Instructions

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

- What is another instruction that has to wait?
- What can we do for that case?

Reordering Instructions

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

- What is another instruction that has to wait?
  - line 8 needs to wait on line 7.
- What can we do for that case?
  - We can move line 9 and line 11 ahead of line 8.

Result of Reordering

line 1: **load R2 address2**
line 2: **load R1 address1**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **store R4 address10**
line 9: **addi R5 R2 30**
line 10: **store R5 address11**
line 11: **add R8 R2 R3**
line 12: **store R8 address12**

line 1 (old 2): **load R1 address1**
line 2 (old 1): **load R2 address2**
line 3 (old 3): **if R1 6**
line 4 (old 4): **addi R3 R1 20**
line 5 (old 5): **goto 7**
line 6 (old 6): **addi R3 R1 10**
line 7 (old 7): **addi R4 R2 5**
line 8 (old 9): **addi R5 R2 30**
line 9 (old 11): **add R8 R2 R3**
line 10 (old 8): **store R4 address10**
line 11 (old 10): **store R5 address11**
line 12 (old 12): **store R8 address12**

line 1: **load R1 address1**
line 2: **load R2 address2**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **addi R5 R2 30**
line 9: **add R8 R2 R3**
line 10: **store R4 address10**
line 11: **store R5 address11**
line 12: **store R8 address12**

Time	Fetch	Decode	Operand Fetch	ALU exec.	Output Save	PC	Notes
1	1	X	X	X	X	1	
2	2	1	X	X	X	2	
3	3	2	1	X	X	3	
4	4	3	2	1	X	4	
5	4	3	X	2	1	4	line 3 waits for line 1 to finish.
6	X	X	3	X	2	4	line 3 moves on. **if** detected. Stop fetching, flush line 4 from fetch step.
7	X	X	X	3	X	4	
8	X	X	X	X	3	4	
9	4	X	X	X	X	4	**if** has finished, PC does **NOT** change.

line 1: **load R1 address1**
line 2: **load R2 address2**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**

line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **addi R5 R2 30**
line 9: **add R8 R2 R3**
line 10: **store R4 address10**

line 11: **store R5 address11**
line 12: **store R8 address12**

Time	Fetch	Decode	Operand Fetch	ALU exec.	Output Save	PC	Notes
9	4	X	X	X	X	4	**if** has finished, PC does **NOT** change.
10	5	4	X	X	X	5	
11	6	5	4	X	X	6	
12	X	X	5	4	X	X	**goto** detected. Stop fetching, flush line 6 from fetch step.
13	X	X	X	5	X	X	
14	X	X	X	X	5	X	
15	7	X	X	X	X	7	**goto** has finished, PC set to 7.
16	8	7	X	X	X	8	
17	9	8	7	X	X	9	

line 1: **load R1 address1**
line 2: **load R2 address2**
line 3: **if R1 6**
line 4: **addi R3 R1 20**
line 5: **goto 7**
line 6: **addi R3 R1 10**
line 7: **addi R4 R2 5**
line 8: **addi R5 R2 30**
line 9: **add R8 R2 R3**
line 10: **store R4 address10**
line 11: **store R5 address11**
line 12: **store R8 address12**

Time	Fetch	Decode	Operand Fetch	ALU exec.	Output Save	PC	Notes
17	9	8	7	X	X	9	
18	10	9	8	7	X	10	
19	11	10	9	8	7	11	
20	12	11	10	9	8	12	
21	X	12	11	10	9	X	
22	X	X	12	11	10	X	
23	X	X	X	12	11	X	
24	X	X	X	X	12	X	
25							program execution has finished!

Execution took 24 clock ticks.
Compare to 31 ticks for the original program.

Review: Example .gdbinit

```
set architecture arm
target remote :1234
symbol-file example.elf
b _start
```

- Sets architecture to arm (default is x86)
- Connects to QEMU process via port 1234
- Loads symbols (labels, etc.) from the ELF file called example.elf
- Puts breakpoint at label _start

Review: GDB Commands

- `b label`
  Sets a breakpoint at a specific label in your source code file. In practice, for some weird reason, the code actually breaks not at the label that you specify, but after executing the next line.
- `b line_number`
  Sets a breakpoint at a specific line in your source code file. In practice, for some weird reason, the code actually breaks not at the line that you specify, but at the line right after that.
- `c`
  Continues program execution until it hits the next breakpoint.
- `i r`
  Shows the contents of all registers, in both hexadecimal and decimal representations; short for `info registers`
- `list`
  Shows a list of instructions around the line of code that is being executed.
- `quit`
  This command quits the debugger, and exits GDB.
- `stepi`
  This command executes the next instruction.
- `set $register=val`
  `set $pc=0`
  This command updates a register to be equal to val, for example, to restart your program, set the PC to 0
- `monitor quit`
  Send the remote monitor (e.g., QEMU in our case) a command, in this case, tell QEMU to terminate; Call this before quiting gdb so that the QEMU process gets killed!

Basic Function Call Example

```
int ex(int g, int h, int i, int j) {
    int f;
    f = (g + h) – (i + j);
    return f;
}

r0 = g, r1 = h, r2 = I, r3 = j, r4 = f
```

Basic Function Call Example Assembly

```
ex:                ; label for function name
SUB sp, sp, #12    ; adjust stack to make room for 3 items
STR r6, [sp,#8]    ; save register r6 for use afterwards
STR r5, [sp,#4]    ; save register r5 for use afterwards
STR r4, [sp,#0]    ; save register r4 for use afterwards

ADD r5,r0,r1     ; register r5 contains g + h
ADD r6,r2,r3     ; register r6 contains i + j
SUB r4,r5,r6     ; f gets r5 – r6, ie: (g + h) – (i + j)
MOV r0,r4        ; returns f (r0 = r4)

LDR r4, [sp,#0] ; restore register r4 for caller
LDR r5, [sp,#4] ; restore register r5 for caller
LDR r6, [sp,#8] ; restore register r6 for caller
ADD sp,sp,#12    ; adjust stack to delete 3 items
MOV pc, lr       ; jump back to calling routine
```

Basic Function Output

```
r0             0xfffffffc       -4
r1             0x4      4
r2             0x6      6
r3             0x7      7
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x10000  0x10000 <_start>
lr             0x1001c  65564
pc             0x1001c  0x1001c <iloop>
cpsr           0x400001d3       1073742291
```

```
@ (g + h) – (i + j)
@ r0 = g
@ r1 = h
@ r2 = i
@ r3 = j
@ r4 = f
        mov r0,#5
        mov r1,#4
        mov r2,#6
        mov r3,#7
        mov r4,#0
```

Basic Function Call Example Stack

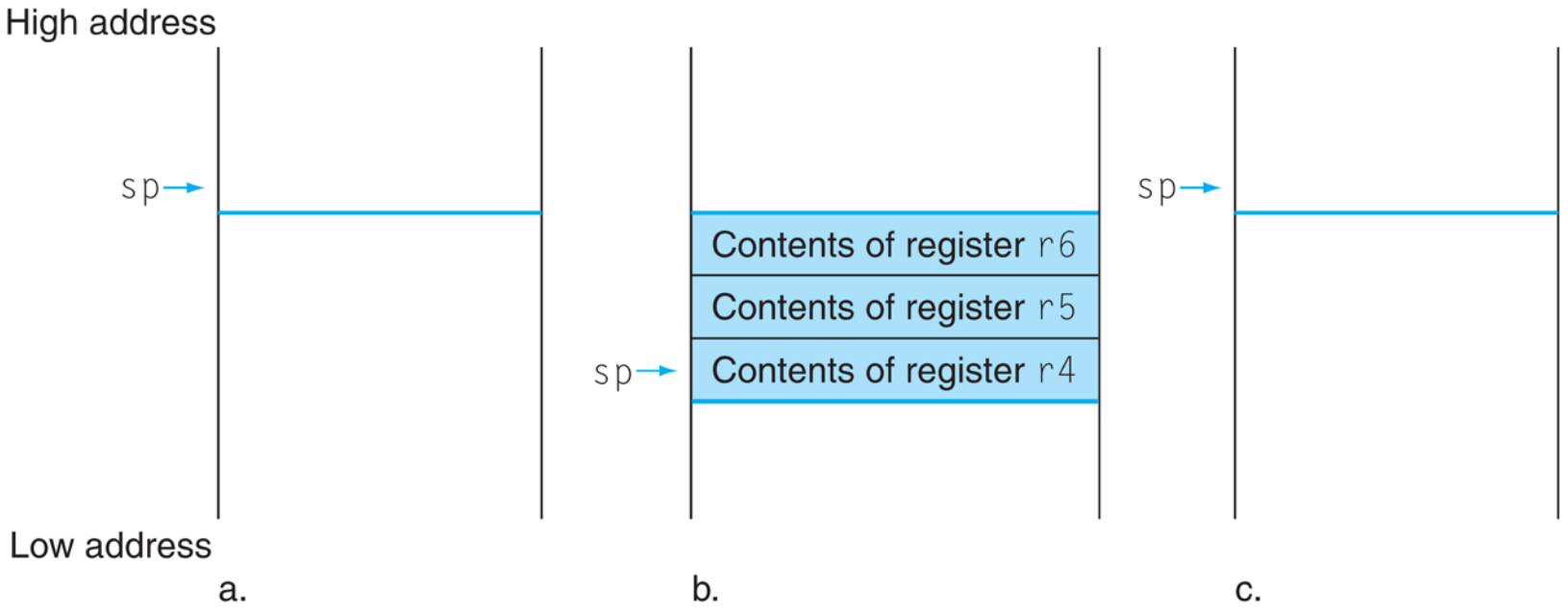


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Recursive Function Example: Factorial

- How do we write function factorial in C, as a recursive function?

```
int factorial(int N)
{
  if (N== 0) return 1;
  return N* factorial(N -1);
}
```

- How do we write function factorial in assembly?

```
@ factorial main body
mov r4, r0
cmp r4, #0
moveq r0, #1
beq factorial_exit

sub r0, r4, #1
bl factorial
mov r5, r0
mul r0, r5, r4
```

Recursive Function Example: Factorial


```
     @ factorial preamble
fact: push {r4,r5,lr}

     @ factorial body
     mov r4, r0
     cmp r4, #0
     moveq r0, #1
     beq fact_exit

     sub r0, r4, #1
     bl fact
     mov r5, r0
     mul r0, r5, r4
```

```
@ factorial wrap-up
fact_exit:
    pop {r4,r5,lr}
    bx lr
```

Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0
(gdb) i r
r0             0x5     5
r1             0x183   387
r2             0x100   256
r3             0x0     0
r4             0x0     0
r5             0x0     0
r6             0x0     0
r7             0x0     0
r8             0x0     0
r9             0x0     0
r10            0x0     0
r11            0x0     0
r12            0x0     0
sp             0xfff4  0xfff4
lr             0x1000c 65548
pc             0x10014 0x10014 <fact+4>
cpsr           0x600001d3      1610613203
```

Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0
(gdb) i r
r0             0x4     4
r1             0x183   387
r2             0x100   256
r3             0x0     0
r4             0x5     5
r5             0x0     0
r6             0x0     0
r7             0x0     0
r8             0x0     0
r9             0x0     0
r10            0x0     0
r11            0x0     0
r12            0x0     0
sp             0xffe8  0xffe8
lr             0x1002c 65580
pc             0x10014 0x10014 <fact+4>
cpsr           0x200001d3      536871379
```

Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0
(gdb) i r
r0             0x3     3
r1             0x183   387
r2             0x100   256
r3             0x0     0
r4             0x4     4
r5             0x0     0
r6             0x0     0
r7             0x0     0
r8             0x0  0
r9             0x0  0
r10            0x0  0
r11            0x0  0
r12            0x0  0
sp             0xffdc     0xffdc
lr             0x1002c    65580
pc             0x10014    0x10014 <fact+4>
cpsr           0x200001d3 536871379
```

Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0
(gdb) i r
r0             0x2     2
r1             0x183   387
r2             0x100   256
r3             0x0     0
r4             0x3     3
r5             0x0     0
r6             0x0     0
r7             0x0     0
r8             0x0  0
r9             0x0  0
r10            0x0  0
r11            0x0  0
r12            0x0  0
sp             0xffd0      0xffd0
lr             0x1002c     65580
pc             0x10014     0x10014 <fact+4>
cpsr           0x200001d3 536871379
```

Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0
(gdb) i r
r0             0x1     1
r1             0x183   387
r2             0x100   256
r3             0x0     0
r4             0x2     2
r5             0x0     0
r6             0x0     0
r7             0x0     0
r8             0x0  0
r9             0x0  0
r10            0x0  0
r11            0x0  0
r12            0x0  0
sp             0xffc4      0xffc4
lr             0x1002c     65580
pc             0x10014     0x10014 <fact+4>
cpsr           0x200001d3 536871379
```

Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0
(gdb) i r
r0             0x0     0
r1             0x183   387
r2             0x100   256
r3             0x0     0
r4             0x1     1
r5             0x0     0
r6             0x0     0
r7             0x0     0
r8             0x0  0
r9             0x0  0
r10            0x0  0
r11            0x0  0
r12            0x0  0
sp             0xffb8      0xffb8
lr             0x1002c     65580
pc             0x10014     0x10014 <fact+4>
cpsr           0x200001d3 536871379
```

Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0
(gdb) i r
r0             0x78    120
r1             0x183   387
r2             0x100   256
r3             0x0     0
r4             0x0     0
r5             0x0     0
r6             0x0     0
r7             0x0     0
r8             0x0  0
r9             0x0  0
r10            0x0  0
r11            0x0  0
r12            0x0  0
sp             0x10000    0x10000 <_start>
lr             0x1000c    65548
pc             0x1000c    0x1000c <iloop>
cpsr           0x600001d3 1610613203
```

Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Stack after final return:

0xff90:     0     0     0     0
0xffa0:     0     0     0     0
0xffb0:     0     0     1     0
0xffc0:     65580 2     0     65580
0xffd0:     3     0     65580 4
0xffe0:     0     65580 5     0
0xfff0:     65580 0     0     65548
0x10000
```

String Output

- So far we have seen character input/output
- That is, one char at a time

- What about strings (character arrays, i.e., multiple characters)?

- Strings are stored in memory at consecutive addresses
  - Like arrays that we saw last time

```
string_abc:
.asciz "abcdefghijklmnopqrstuvwxyz\n\r"
.word 0x00
```

ADDR	Byte 3	Byte 2	Byte 1	Byte 0
0x1000	'd'	'c'	'b'	'a'
0x1004	'h'	'g'	'f'	'e'
0x1008	'l'	'k'	'j'	'i'
0x100c	'p'	'o'	'n'	'm'
0x1010	't'	's'	'r'	'q'
0x1014	'x'	'w'	'v'	'u'
0x1018	'\r'	'\n'	'z'	'y'

Assembler Output

```
0001012e <string_abc>:
   1012e: 64636261   strbtvs   r6, [r3], #-609; 0x261
   10132: 68676665   stmdavs   r7!, {r0, r2, r5, r6, r9, sl, sp,
lr}^
   10136: 6c6b6a69   stclvs    10, cr6, [fp], #-420; 0xfffffe5c
   1013a: 706f6e6d   rsbvc     r6, pc, sp, ror #28
   1013e: 74737271   ldrbtvc   r7, [r3], #-625; 0x271
   10142: 78777675   ldmdavc   r7!, {r0, r2, r4, r5, r6, r9, sl,
ip, sp, lr}^
   10146: 0d0a7a79   vstreq    s14, [sl, #-484]    ; 0xfffffe1c
   1014a: 00000000   andeq     r0, r0, r0
```

Printing Strings

```
@ assumes r0 contains uart data register address
@ r1 should contain address of first character of string
@ to display; stop if 0x00 ('\0') seen
print_string: push  {r1,r2,lr}
str_out: ldrb  r2,[r1]
     cmp  r2,#0x00  @ '\0' = 0x00: null character?
     beq  str_done  @ if yes, quit
     str  r2,[r0]   @ otherwise, write char of string
     add  r1,r1,#1  @ go to next character
     b    str_out   @ repeat
str_done: pop  {r1,r2,lr}
     bx   lr
```