SEC204

Computer Architecture and Low Level Programming

Dr. Vasilios Kelefouras

Email: v.kelefouras@plymouth.ac.uk

Website:
https://www.plymouth.ac.uk/staff/vasilios-kelefouras

Introduction

Outline

- Superscalar processors
- Superpipelining processors
- In order and out of order processors
- RISC, CISC, VLIW and EPIC processors
- Moore's Law

Superscalar Processors

- You have been taught about CPU pipelining but performance is never enough
- Any other options?

- **Why not make the pipeline deeper and deeper?**
  - By adding more pipeline stages the clock cycle is reduced
  - But beyond some point, adding more pipe stages doesn't help, because control/data hazards increase, and become costlier

Superscalar Processors

- A superscalar processor can execute more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to different execution units on the processor
- Duplicates the pipeline to accommodate **Instruction Level Parallelism** (ILP)
  - Note that duplicating HW in just one pipe stage doesn't help, e.g., when having 2 ALUs, the bottleneck moves to other stages
- It therefore achieves more throughput (the number of instructions that can be executed in a unit of time) than would otherwise be possible at a given clock rate
- Superscalar machines issue a variable number of instructions each clock cycle, up to some maximum
  - **instructions must be independent – no data or control dependencies**

Pipeline vs 2 way Superscalar (pipelined)

	1	2	3	4	5	6	7	8
Instruction1	IF	ID	EX	MEM	WB			
Instruction2		IF	ID	EX	MEM	WB		
Instruction3			IF	ID	EX	MEM	WB	
Instruction4				IF	ID	EX	MEM	WB

Instruction1	IF	ID	EX	MEM	WB		
Instruction2	IF	ID	EX	MEM	WB		
Instruction3		IF	ID	EX	MEM	WB	
Instruction4		IF	ID	EX	MEM	WB	

Careful: If the instructions are interdependent, then superscalar does not improve performance at all

Superscalar Hardware Support

- Extra hardware to simultaneously fetch multiple instructions is needed
- Hardware logic to determine data dependencies is needed, involving the registers' values
- Extra hardware (functional units) to execute multiple instructions in parallel
- Extra hardware (functional units) to issue multiple instructions in parallel
- More extra hardware for more complicated notions (out of the scope of this module)

- Extra Performance does not come for free

Think Pair Share

Describe the pipeline stages of the following code for a) a 5 stage pipelined processor and b) 5 stage superscalar pipelined processor with 2 pipes. Compare the number of cycles

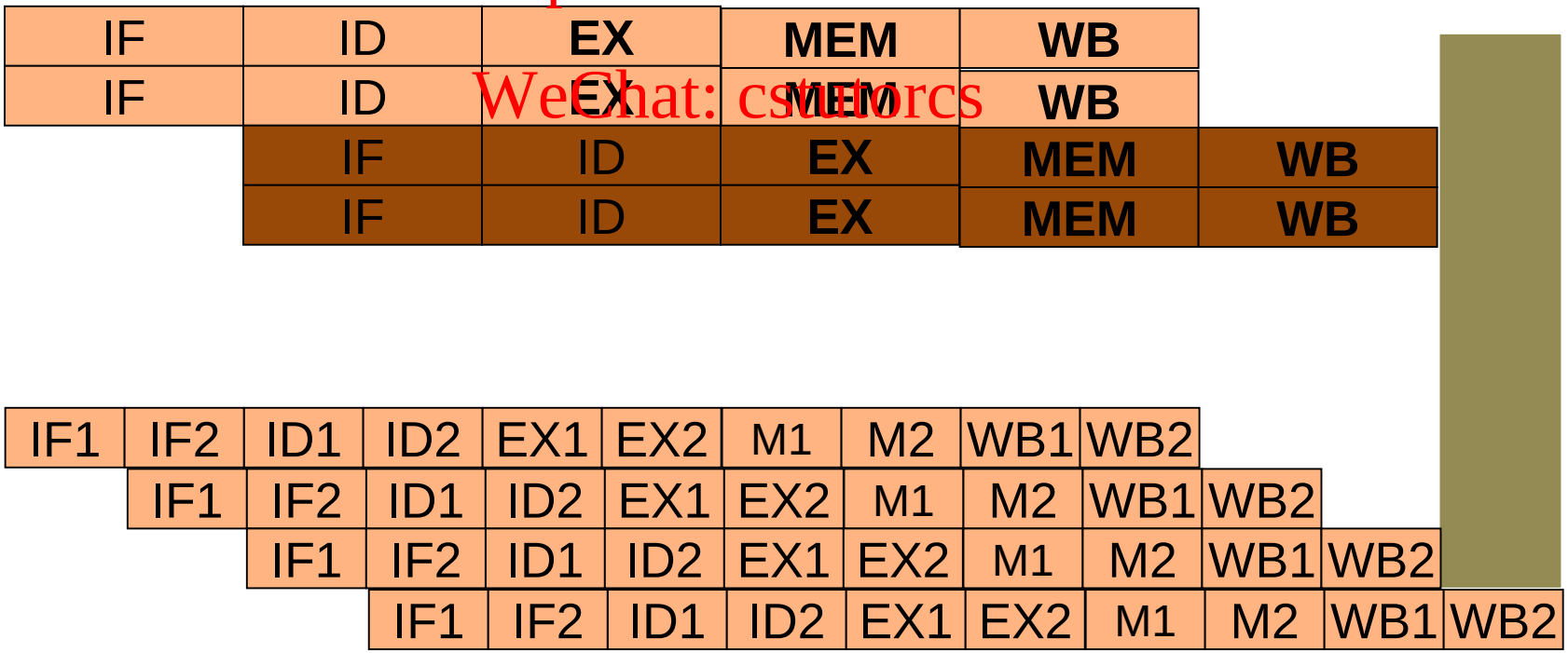
	1	2	3	4	5	6	7	8	9
Instruction1	IF	ID	EX	MEM	WB				
Instruction2		IF	stall	ID	EX	MEM	WB		
Instruction3			stall	IF	ID	EX	MEM	WB	
Instruction4				stall	IF	ID	EX	MEM	WB

Instruction1	IF	ID	EX	MEM	WB			(1st pipe)
Instruction2		IF	stall	ID	EX	MEM	WB	(1st pipe)
Instruction3		IF	ID	EX	MEM	WB		(2nd pipe)
Instruction4			IF	ID	EX	MEM	WB	(2nd pipe)

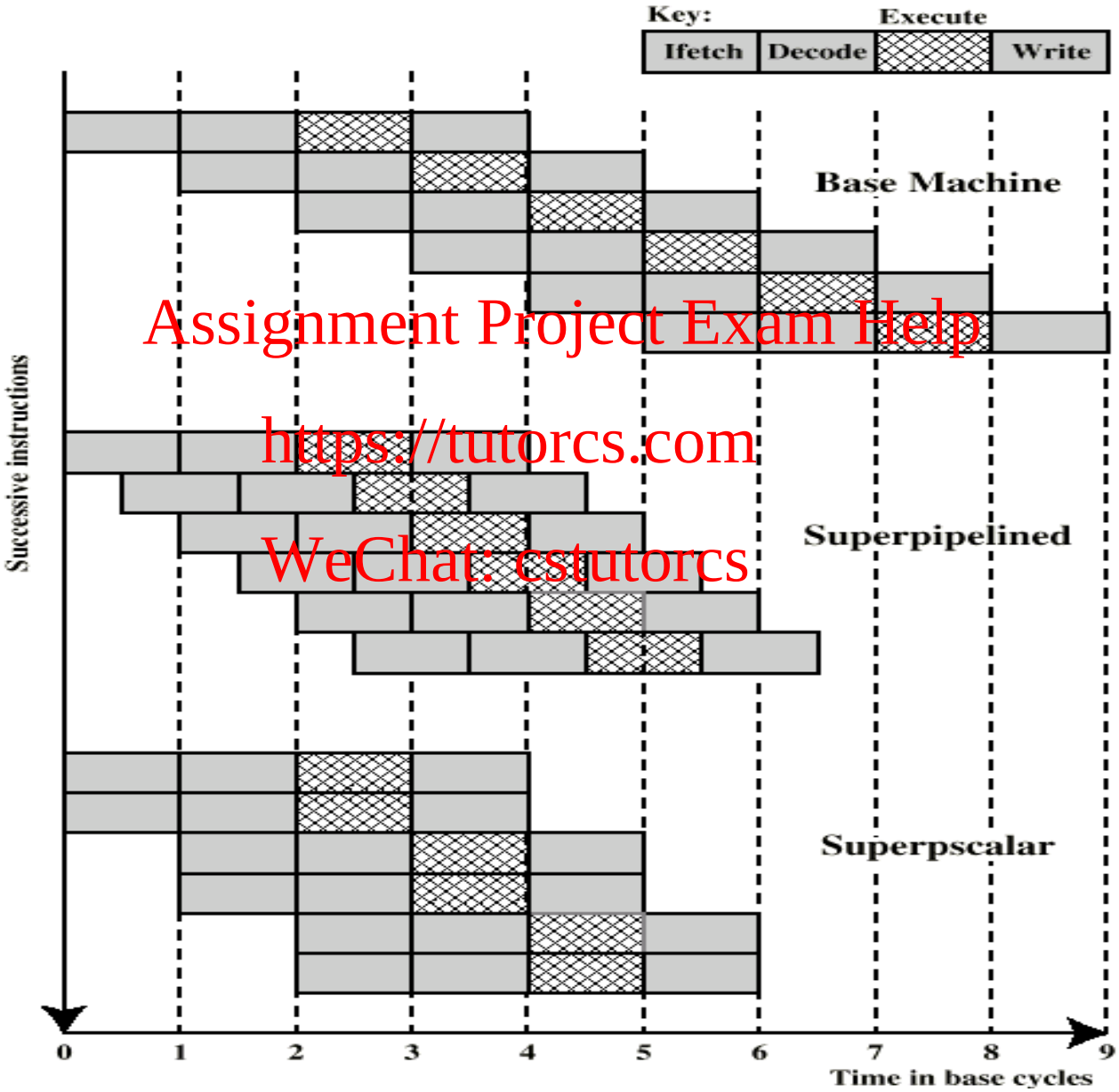
IMUL R3 ← R1, R2
ADD R3 ← R3, R1
IMUL R5 ← R6, R8
ADD R7 ← R6, R8

Superscalar vs Superpipelining (1)

- **Superpipelining:** *Vaguely defined as deep pipelining, i.e., lots of stages*
- **Superscalar issue complexity:** limits super-pipelining
- How to compare them?
  - e.g., 2-way Superscalar vs. two stages



Superscalar vs Superpipelining (2)



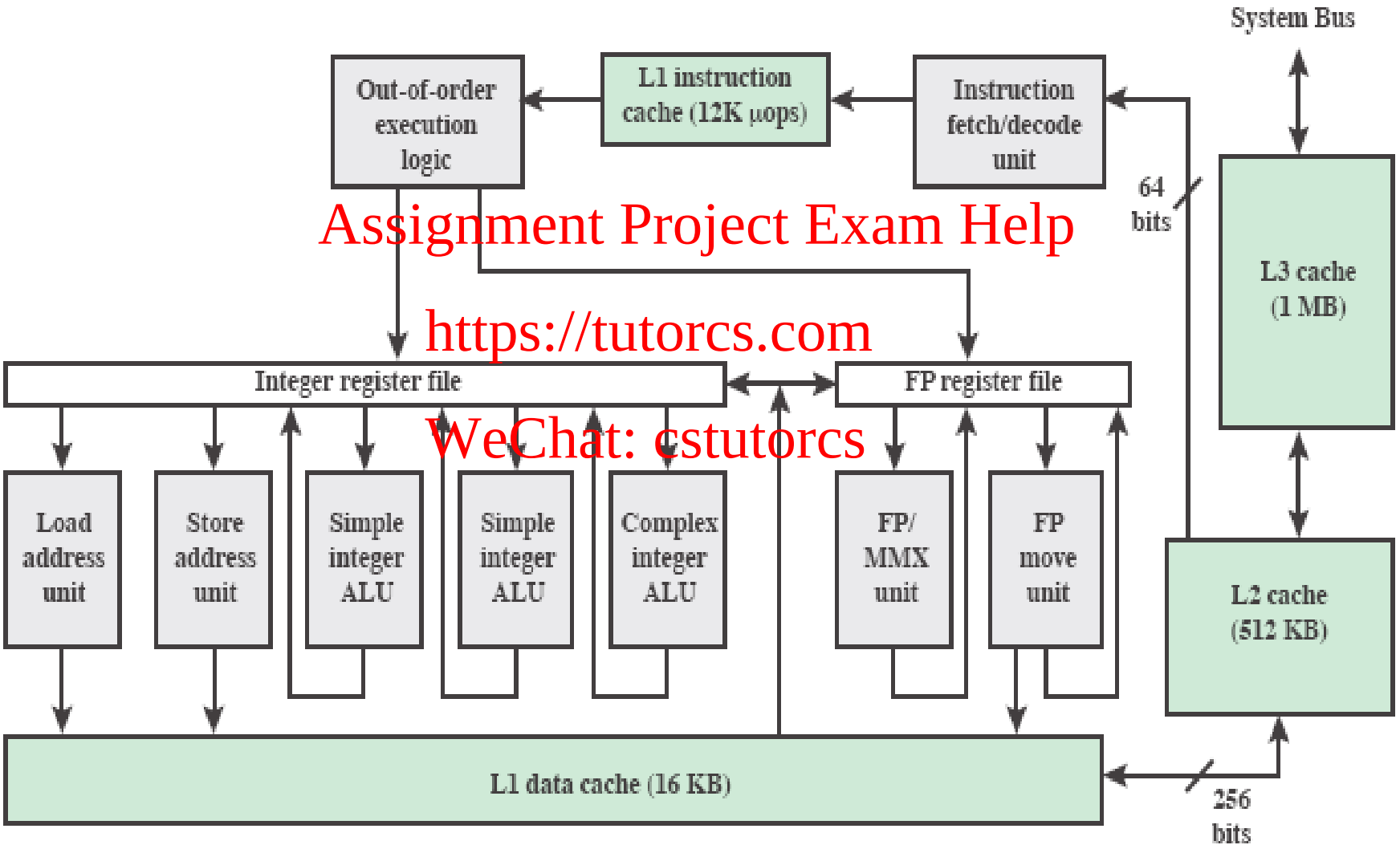
Superscalar Problem

- what if two successive instructions can't be executed in parallel?
  - **Superscalar operation is double impacted by a stall**

- Superscalar depends upon:
  - Instruction level parallelism (ILP)
  - Compiler based optimization

- Limited by
  - Data dependencies
  - Control dependencies

A Superscalar Processor



Is superscalar good enough?

- **A superscalar processor can fetch, decode, execute more than one instructions in parallel**
- But…
  - Can execute only independent instructions in parallel
    - Whereas adjacent instructions are often dependent
  - **So the utilization of the second pipe is often low**
- **Solution: out-of-order execution**
  - Execute independent instructions in a different, more efficient order
  - A specific HW mechanism examines a sliding window of consecutive instructions (**instruction window** – it is a small memory)
  - Ready instructions get picked up from window and executed out of order
  - Instructions enter (**dispatched**) and leave (**committed**) the instruction window in program order, and an instruction can only leave the window when it is the oldest instruction in the window and it has been completed

Out of Order Processors (1)

- **The pipelines we have studied so far are statically scheduled and inorder**, i.e., instructions are executed in program's order
- If a hazard causes stall cycles, then all instructions up to the offending instruction are stalled
  - Forwarding, branch prediction, and other techniques can reduce the number of stall cycles we need, but sometimes a stall is unavoidable

- **Consider the following assembly code in a superscalar processor**
  - The second instruction stalls the second pipe

```
IMUL  R3 ← R1, R2
ADD   R3 ← R3, R1
IMUL  R5 ← R6, R8
ADD   R7 ← R6, R8
```

- **What about changing the order of the instructions?**
  - Careful: data dependencies must be preserved

```
IMUL  R3 ← R1, R2
IMUL  R5 ← R6, R8
ADD   R7 ← R3, R5
ADD   R3 ← R3, R1
```

Data flow analysis

S1: r1=r0/7 //*division needs many cycles*

S2: r8=r1+r2

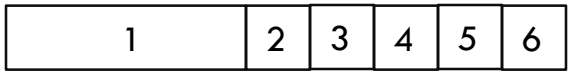
S3: r5=r5+1

S4: r6=r6-r3

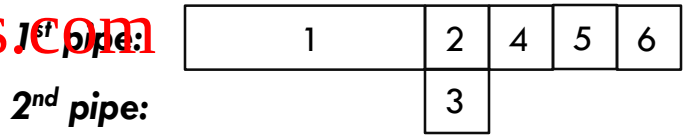
S5: r4=r5+r6

S6: r7=r8*r4

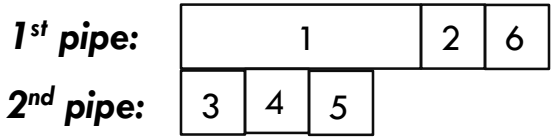
In order execution:



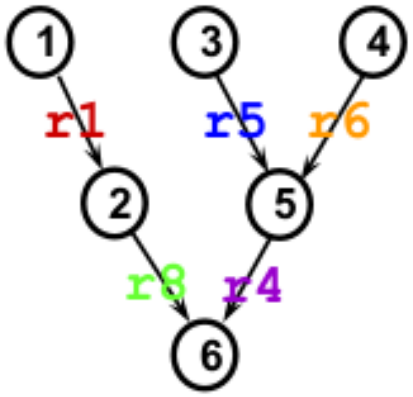
In order execution 2-way superscalar:



Out of order execution 2-way superscalar:



Data Flow Graph



Out of Order Processors (2)

- **Idea: Move the dependent instructions out of the way of independent ones**
  - Rest areas for dependent instructions: **Reservation station**
  - Monitor the source "values" of each instruction in the resting area
  - When all source "values" of an instruction are available, dispatch the instruction
  - Allows independent instructions to execute and complete in the presence of a long latency operation

Out of Order Processors (3)

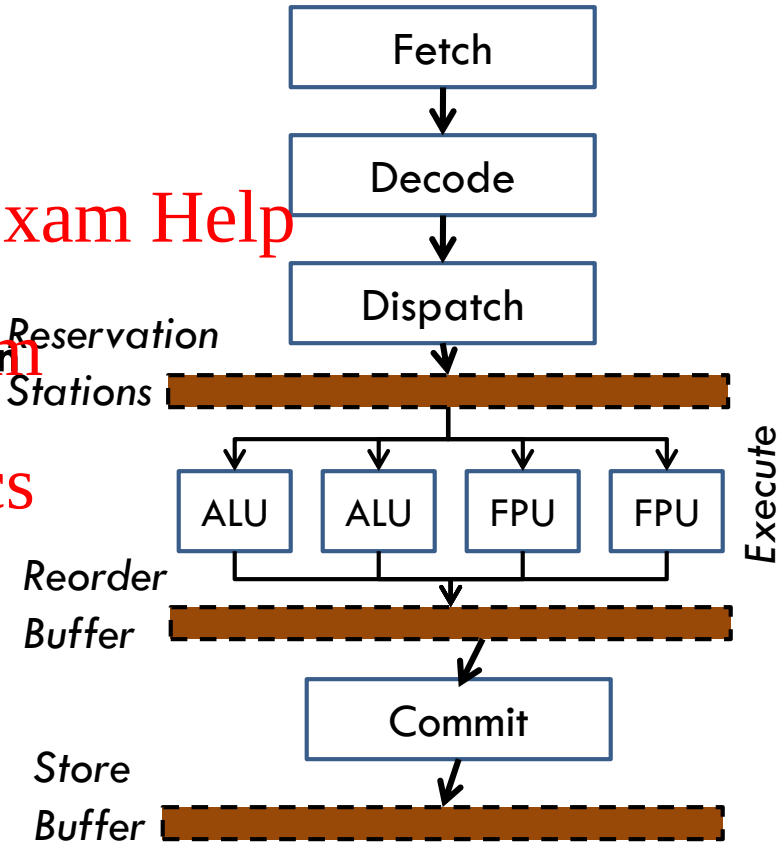
- **Instruction window**: It is a memory that holds the instructions that have been fetched and decoded and are waiting to be executed
  - Note: Often, the instruction window doesn't actually exist as a single buffer, but is distributed among reservation stations

- **Enhanced issue logic.** The issue logic must be enhanced to issue (start) instructions out of order depending on their readiness to execute

- **Reservation stations.** Each functional unit has a set of reservation stations associated with it. Each station contains information about instructions waiting or ready to issue.

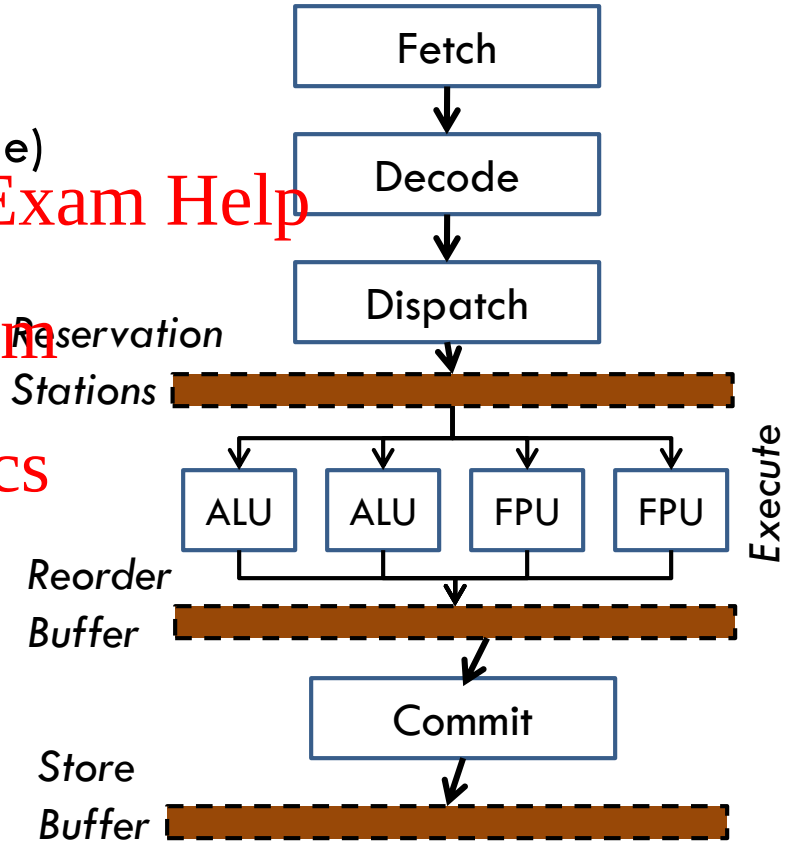
Out of Order Execution (1)

- **Fetch, decode & dispatch instructions in order**
  - Multiple instructions are fetched/decoded/dispatched in parallel
  - Instructions are put in the instruction window – reservation stations (RS)
- **Execute instructions (out of order) that are ready in the reservation stations – speculative execution**
  - Instruction operands must be ready
  - Available execution resources
- After execution:
  - Broadcast result on bypass network
  - Signal all dependent instructions that their data are ready
- Commit instructions **in-order**
  - All execution within the instruction window is speculative (i.e., side-effects are not applied outside the CPU) until it is committed



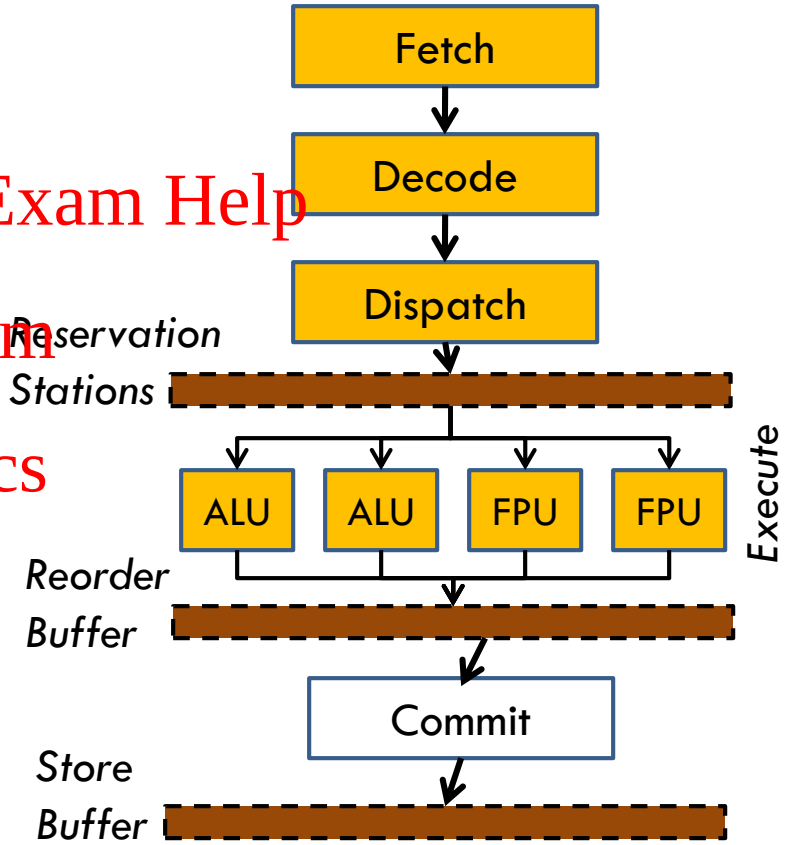
Out of Order Execution (2)

- **Advantages: Better performance!**
  - Exploit Instruction Level Parallelism (ILP)
  - Hide latencies (e.g., L1 data cache miss, divide)
- **Disadvantages:** HW is much more complex than that of in-order processors
  - **More expensive, larger chip area, higher power consumption**
- Can compilers do this work instead?
  - In a very limited way
  - Compilers lack runtime information
    - Conditional branch direction
    - Data values, which may affect calculation time and control
    - Cache miss / hit



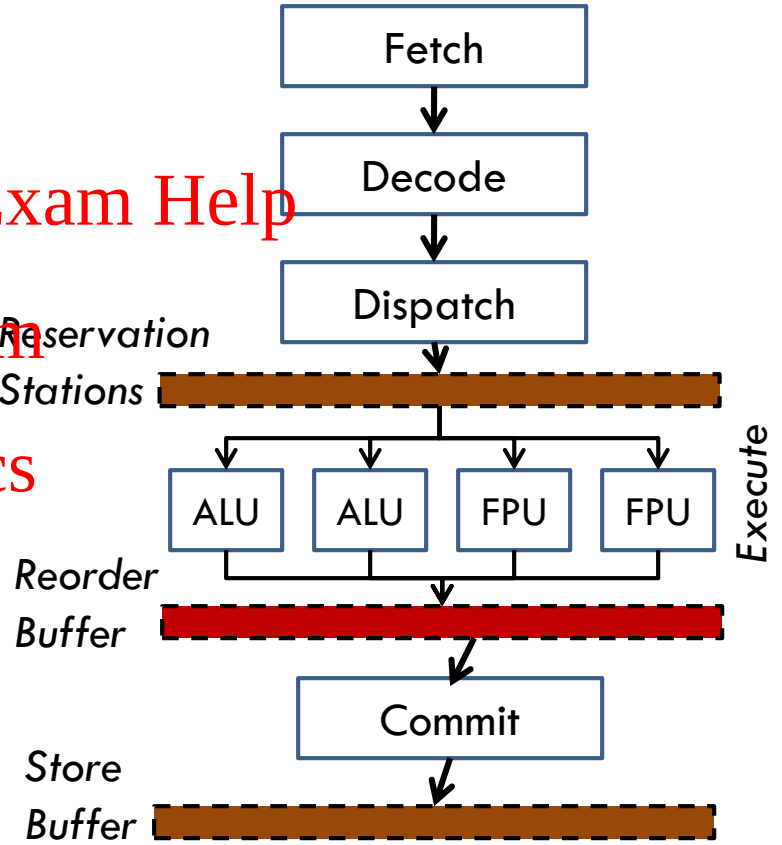
Out of Order Processors – the Pipeline (1)

- **Fetch**
  - Branch prediction
- **Decode**
  - Register renaming (see next)
- **Dispatch** : new instructions are added to the instruction window - reservation stations (RS)
- **Reservation stations (RS)**
  - Instructions wait for their inputs
  - Instructions wait for their functional units
  - If instruction operands are ready they are sent to the FU
  - Otherwise, check on bypass network and wait for operands
- Functional units (FU)
  - ALUs, AGUs, FPUs



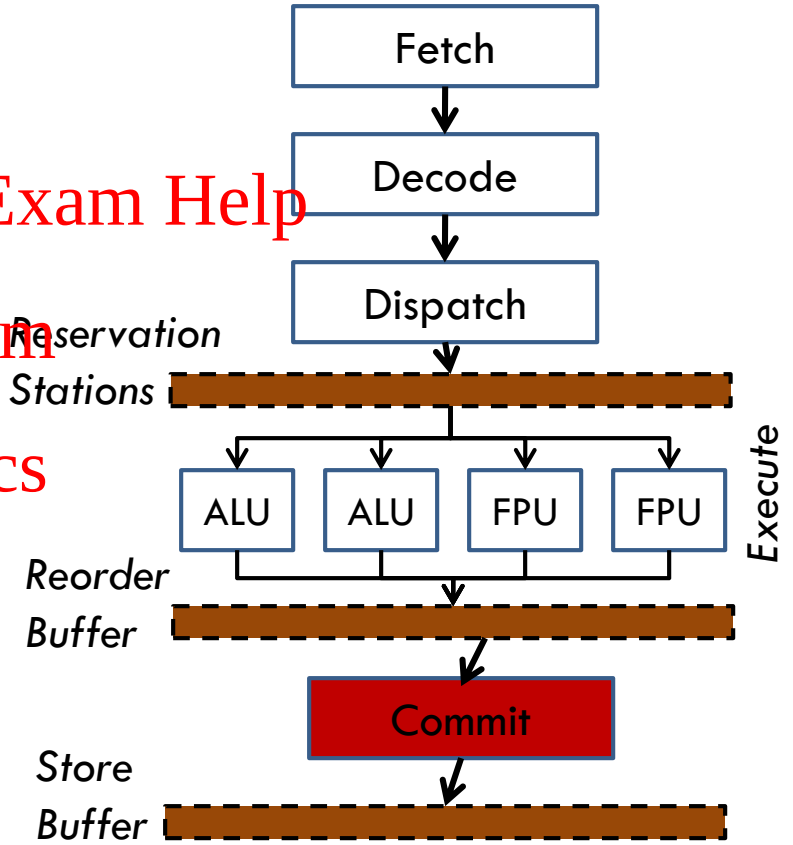
Out of Order Processors – the Pipeline (2)

- **Bypass network**
  - Broadcast computed values back to reservation stations
- **ReOrder buffer (ROB)**
  - It allows instructions to be committed in-order
  - De-speculates execution, mostly by Committing instructions in-order
  - flushes the speculative instructions when a misprediction is discovered



Out of Order Processors – the Pipeline (3)

- **Commit**
  - All execution within the instruction window is speculative (i.e., side-effects are not applied outside the CPU) until it is committed
  - Instructions can write to memory only once it is certain they should have been executed
  - Instructions must not affect machine state while they are speculative
  - Instructions enter and leave the instruction window in program order, and an instruction can only leave the window when it is the oldest instruction in the window and it has been completed

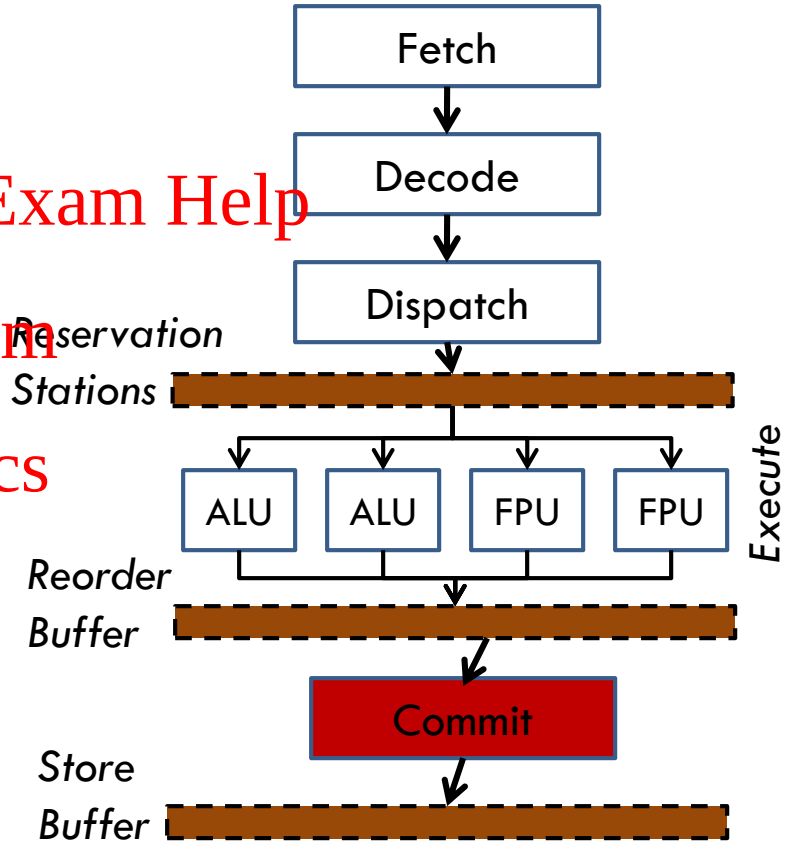


- **The instruction window is instantiated as RS & ROB**

Out of Order Processors – the Pipeline (4)

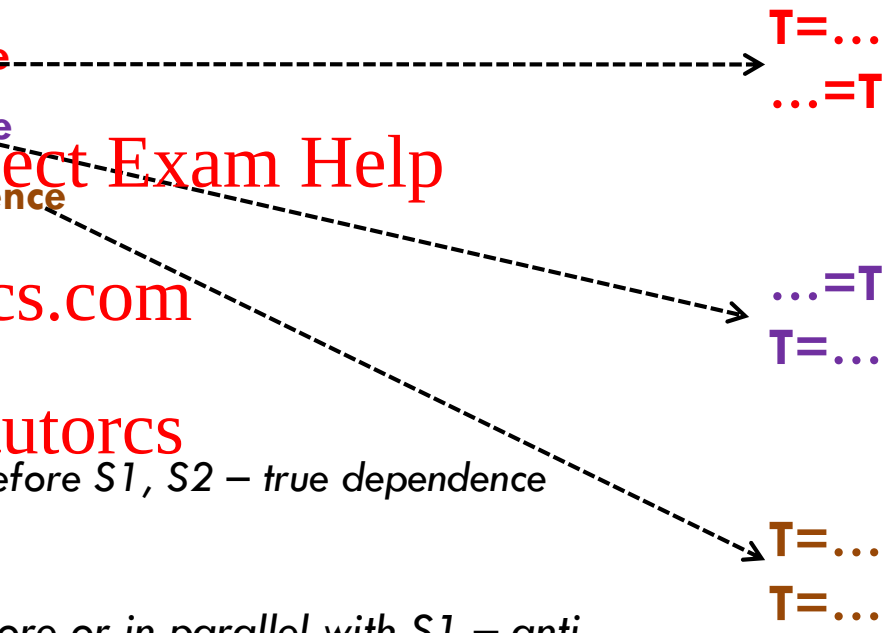
Store Buffer:

- **Stores are not executed OoO**
  - Stores are never performed speculatively
  - There is no transparent way to undo them
- Stores are also never re-ordered among themselves
  - The Store Buffer dispatches a store only when
    - The store has both its address and its data ready and
    - There are no older stores awaiting dispatch



Data Dependencies and Register Renaming

- **Data dependencies reside into 3 categories**
  - **Read after Write (RAW) or true dependence**
  - **Write after Read (WAR) or anti-dependence**
  - **Write after Write (WAW) or output dependence**



A:

```
S1: PI=3.14;
S2: R=2;
S3: S=2 x PI x R    //S3 cannot be executed before S1, S2 – true dependence
```

B:

```
S1: T1=R1;      //S3 cannot be executed before or in parallel with S1 – anti-
S2: R2=PI-T1;   //dependence. But it can be eliminated by applying register
S3: R1=PI+S;    //renaming
```

```
S1: T1=R1;
S2: R2=PI-T1;
S3: R3=PI+S;
```

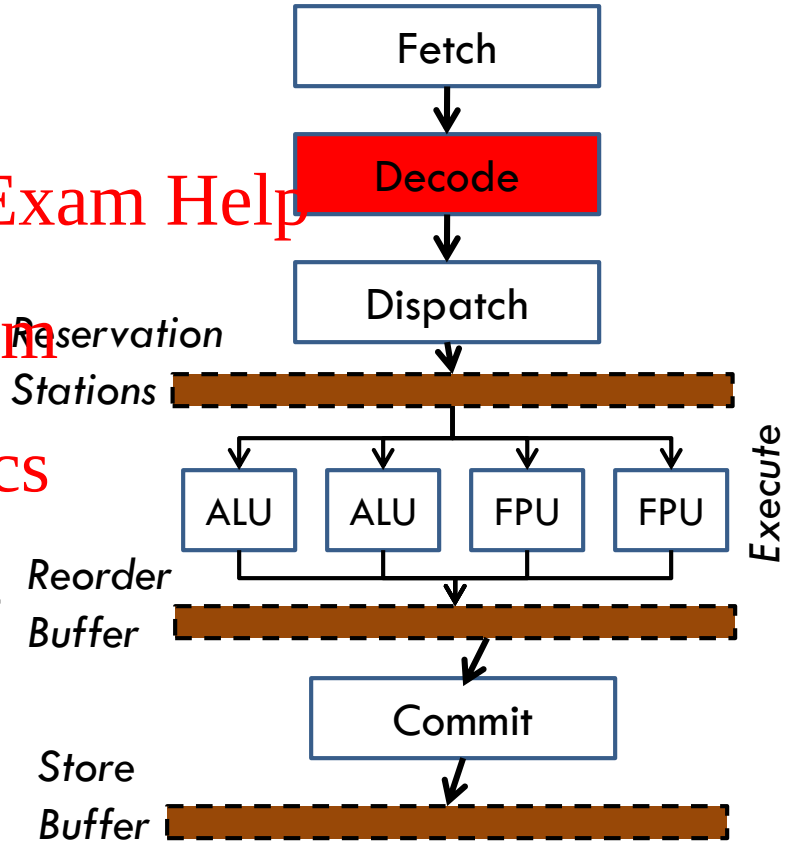
C:

```
S1: T1=R1;        S1: T1=R1;
S2: T1=R2+5;  ⇨   S2: T2=R2+5;
```

WAW dependence is eliminated by applying register renaming

Register Renaming

- Register renaming is a technique that eliminates the false data dependencies
- Changes register names to eliminate WAR/WAW hazards
- The elimination of these false data dependencies reveals more ILP
- **However, True dependencies cannot be eliminated**
- Register renaming is a new pipeline stage that allocates physical/hardware registers to instances of logical registers in the **decode stage**



Register Renaming – example (1)

D = (a+b) * (a+b-c) //high level code

Before Register Renaming:

I1: load r1,a
I2: load r2,b
I3: r3=r1+r2
I4: load r1,c
I5: r2=r3-r1
I6: r1=r3*r2
I7: st address,r1

Its assembly code

Draw its Data Flow Graph…

After Register Renaming:

I1: load r1,a
I2: load r2,b
I3: r3=r1+r2
I4: load r4,c
I5: r5=r3-r4
I6: r6=r3*r5
I7: st address,r6

Draw its Data Flow Graph…

Now, speaking about performance, is it better?

Register Renaming – example (2)

After Register Renaming:

```
I1:   load  r1,a
I2:   load r2,b
I3:   r3=r1+r2
I4:   load r4,c
I5:   r5=r3-r4
I6:   r6=r3*r5
I7:   st addres,r6
```

In this code, 4 pipeline stalls occur.

Solution: Reorder instructions as follows – **or equally, execute out of order**

After OoO:

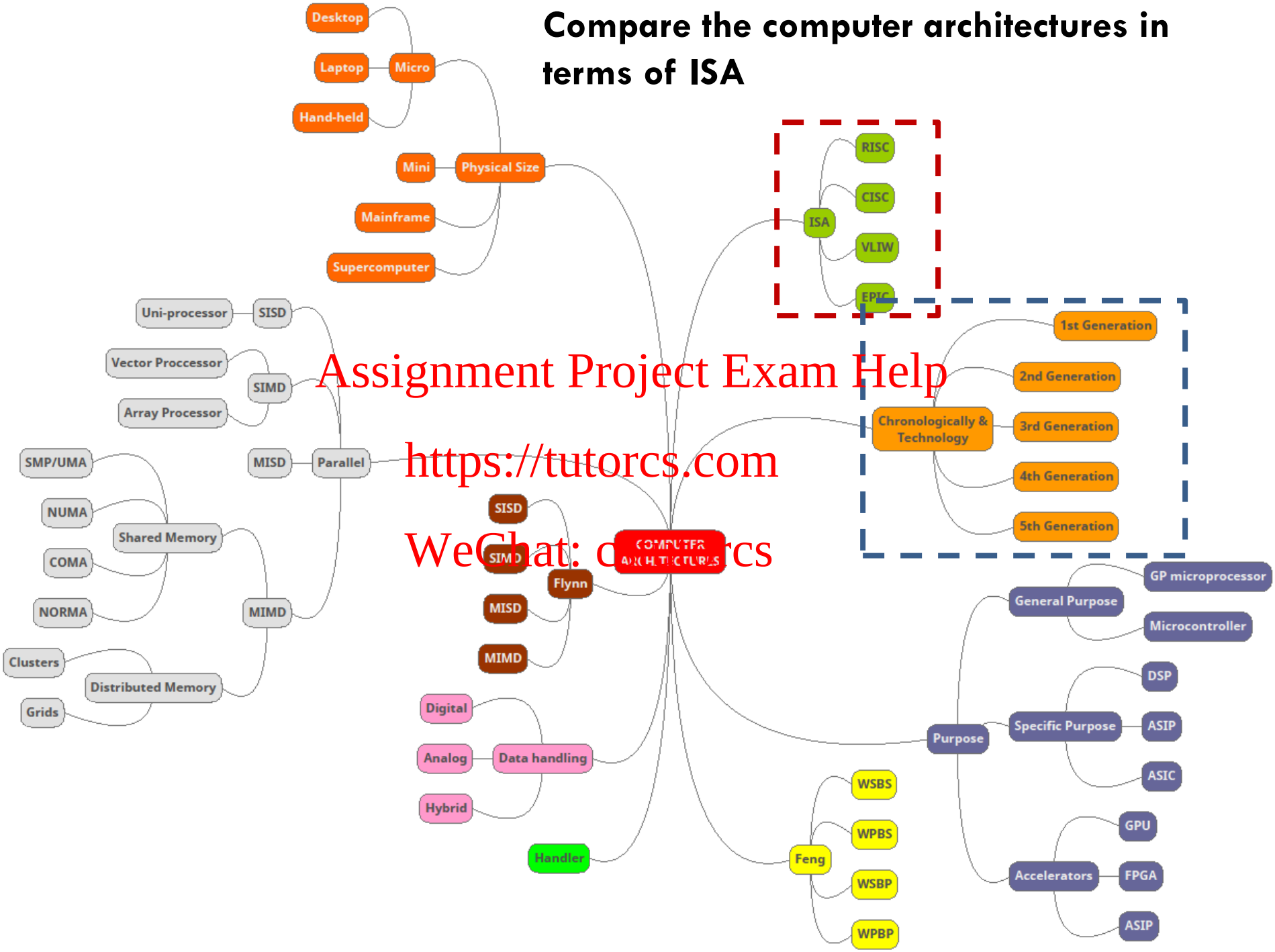
```
I1:       load  r1,a
I2:       load r2,b
I4:       load r4,c  -> hiding the latency from i2 to i3..
I3:       r3=r1+r2
I5:       r5=r3-r4
I6:       r6=r3*r5
I7:       st addess,r6
```

***Conclusion**: In this code, no stalls occur, since none of the load instructions is immediately followed by a dependent (arithmetic) instruction*

Out of Order Processors - Summary

- Advantages
  - Help exploit Instruction Level Parallelism (ILP)
  - Help hide latencies (e.g., cache miss, divide)
  - Superior/complementary to instuction Scheduler in the compiler
    - Dynamic instruction window
- Complex micro-architecture - hardware
  - Complex instruction logic
  - Requires reordering mechanism (retire/commit)
  - Misprediction/speculation recovery
- Speculative Execution
  - Advantage: larger scheduling window ⇒ reveals more ILP
  - Issues:
    - Complex logic needed to recover from mis-prediction
    - Runtime cost incurred when recovering from a mis-prediction

Compare the computer architectures in terms of ISA



What is ISA (Instruction Set Architecture)

- It provides commands to the processor, to tell it what to do, e.g., add, load, store
- ISA is analogous to a human language
- Allows communication
  - Human language: person to person
  - ISA: software to hardware
- Need to speak the same language/ISA
- Many common aspects
  - Part of speech: verbs, nouns, adjectives, adverbs, etc.
  - Common operations: calculation, control/branch, memory
- Different computer processors may use almost the same instruction set

RISC vs CISC (1)

- Complex instruction set computer (**CISC**): complex instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store)
  - CISC puts emphasis on HW
  - CISC was developed to make compiler development simpler
  - CISC is typically used for **general purpose computers**
- Reduced instruction set computer (**RISC**): simple and 1 cycle instructions
  - RISC puts emphasis on SW
  - RISC was developed to make HW simpler
  - RISC is typically used for **smart phones, tablets and other embedded devices**

RISC vs CISC (An example)

❑ Let's say we want to find the product of two numbers – (1,3) and (2,1) and then store the product back in the location (1,3)

C code: *A[1][3]=A[1][3]*A[2][1];*

1. **CISC Approach:** ***MULT $(1,3), $(2,1)***
   - *Complex instruction*
   - *HW will do most of the work*
2. ***RISC Approach:***

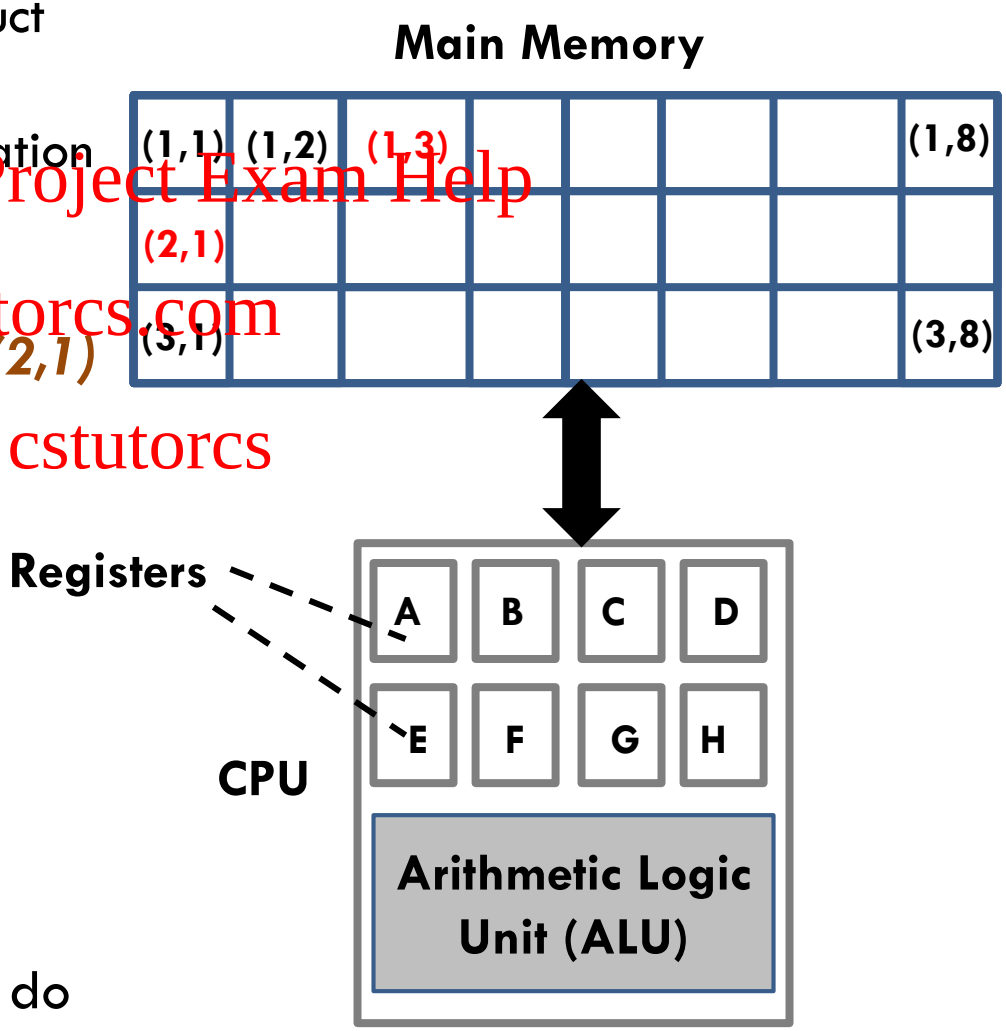
   LOAD A, $(1,3)

   LOAD B, $(2,1)

   MUL A, B

   STORE $(1,3), A

   - *4 simple instructions*
   - The compiler has more work to do



RISC vs CISC architecture comparison

CISC

1. Instructions take varying amount of cycle time (multiple cycles)
2. Instructions provide complex operations
3. Many different instructions
4. Less registers
5. Pipeline is difficult
6. Different length instructions
7. The Opcode has no fixed position and size

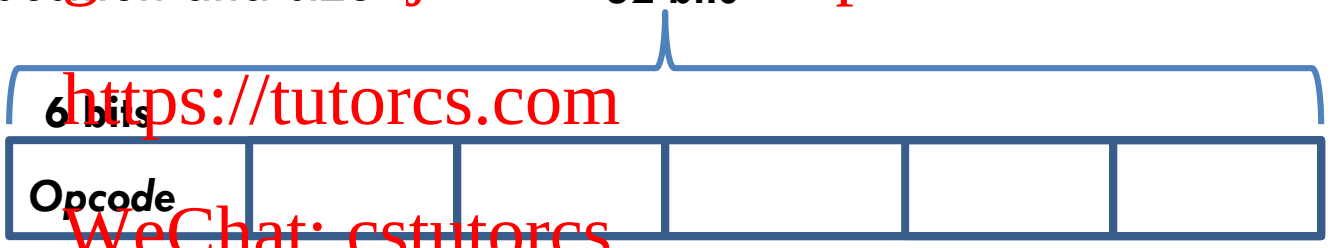
RISC

1. 1 cycle instruction
2. Simple operations
3. Few different instructions
4. More registers
5. Pipeline is easy
6. All instructions have the same length(4 bytes)
7. The Opcode has a fixed position and size within the instruction

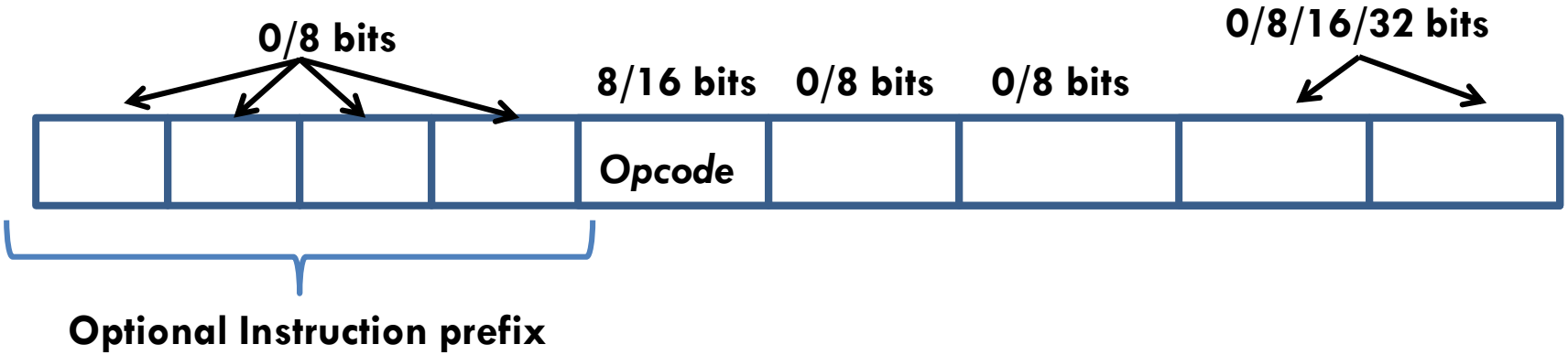
RISC vs CISC (2)

- Opcode specifies the operation to be performed
  - RISC – fixed position and size
  - CISC – no fixed position and size

RISC instruction format:



CISC instruction format:



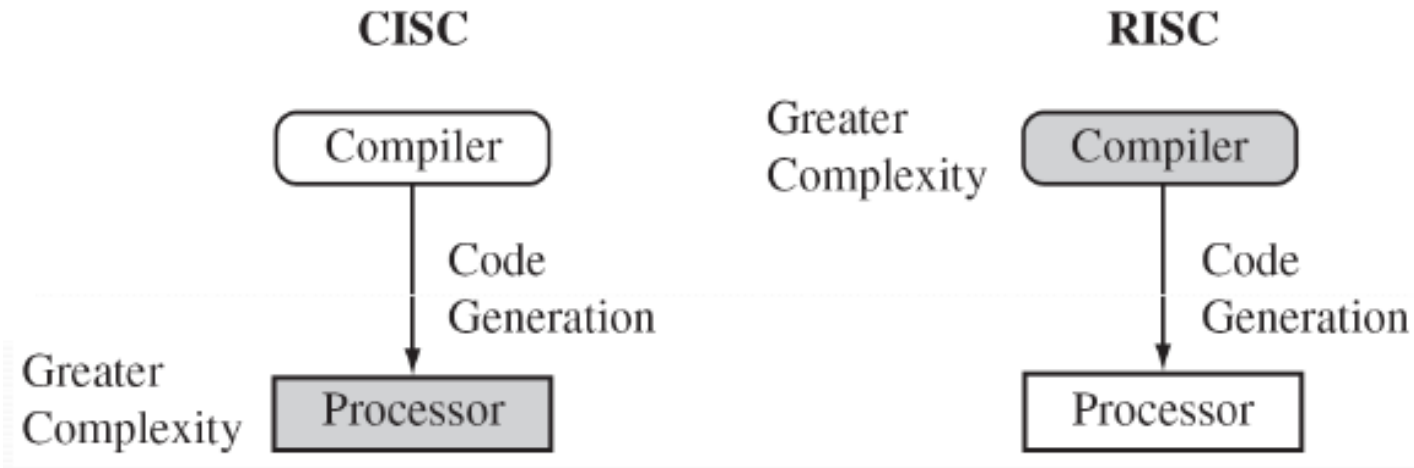
RISC vs CISC - Pros & Cons

CISC:

- Emphasis on HW
- Multi-clock complex instructions
- Less "instructions/program" with "complex" instructions
- Easy for assembly-level programmers, good code density
- Easy for compiler writers support more complex higher-level languages

RISC:

- Emphasis on SW
- Single-clock simple instructions
- Less "cycles/instruction" with single-cycle instructions
- Faster design and implementation - RISC takes about 1/5 the design time
- The performance of the RISC processors highly depends mostly on the compiler or programmer



RISC vs CISC - Conclusions

- **Nowadays, the two architectures almost seem to have adopted the strategies of the other**
- **CISC and RISC implementations are becoming more and more alike**
  - Today's RISC chips
    - support as many instructions as yesterday's CISC chips
    - incorporate more complicated, CISC-like commands
    - make use of more complicated hardware, making use of extra function units for superscalar execution
  - Today's CISC chips
    - use many techniques formerly associated with RISC chips
    - are now able to execute more than one instructions within a single clock

VLIW (Very Large Instruction Word) Processors (1)

- Multiple function units execute all operations in a large instruction concurrently
- A fixed number of operations is formatted as one big instruction (called a bundle)
- Fixed format so could decode operations in parallel
- VLIW CPUs use SW (the compiler) and not HW to decide which operations can run in parallel in advance
  - complexity of hardware is reduced substantially
- However, branch miss-prediction or cache misses may stall the processor
- Compiler does the work of the missing HW
- **Normally, they are used as digital signal processors (DSPs)**
- VLIW is a lot simpler than superscalar designs, but has not so far been commercially successful

VLIW (Very Large Instruction Word) Processors (2)

A problem with traditional VLIW is code size

- Often it is simply not possible to completely utilize all processor execution units
- Thus many instructions contain NOPs in portions of the instruction word with a corresponding **increase in the size of the code**
- **Increased code size has obvious implications for the efficacy of caches and bus bandwidth**
- **Modern VLIWs deal with this problem in different ways**
  - One simple method is to offer several instruction templates, and allow the compiler to pick the most appropriate one – in this case, the one that utilizes the most bits of the instruction word
  - Another is to employ traditional data-compression techniques to the code

VLIW vs (RISC & CISC)

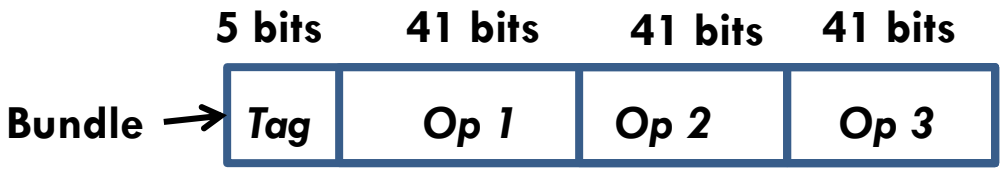
VLIW processors have the following benefits over RISC & CISC:

- Faster
- Lower power consumption
- Reduced HW complexity
- Cheaper
- Reduced design time

VLIW Drawbacks:

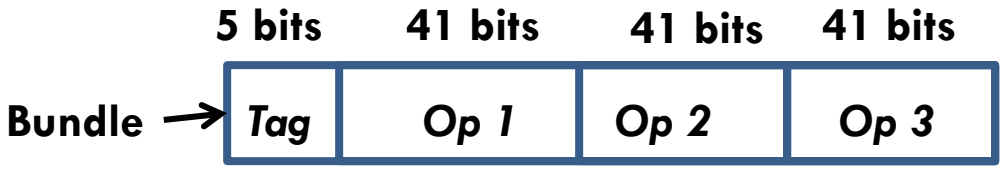
- No code compatibility between different generations
- Large code size - Empty slots are filled with NOPs
- Compilers are extremely complex
- Parallelism is underutilized for some algorithms – dependencies introduce NOPs

EPIC (explicitly parallel instruction computer)



- Very closely related to but not the same as VLIW
- Combines the best attributes of VLIW & superscalar RISC
- So far the only implementation of this architecture is as part of the IA-64 processor architecture in the **Intel Itanium** family of processors
- EPIC instruction word contains three 41-bit instructions and a 5-bit control field
- Applicable to general-purpose computers
- Compiler must find or create sufficient ILP
- Compiler gather instructions in groups
  - All the instructions in a group can be executed in parallel

EPIC – Pros & Cons



- Benefits over VLIW
  - Code size is reduced
  - The same code can be executed on different processor implementations

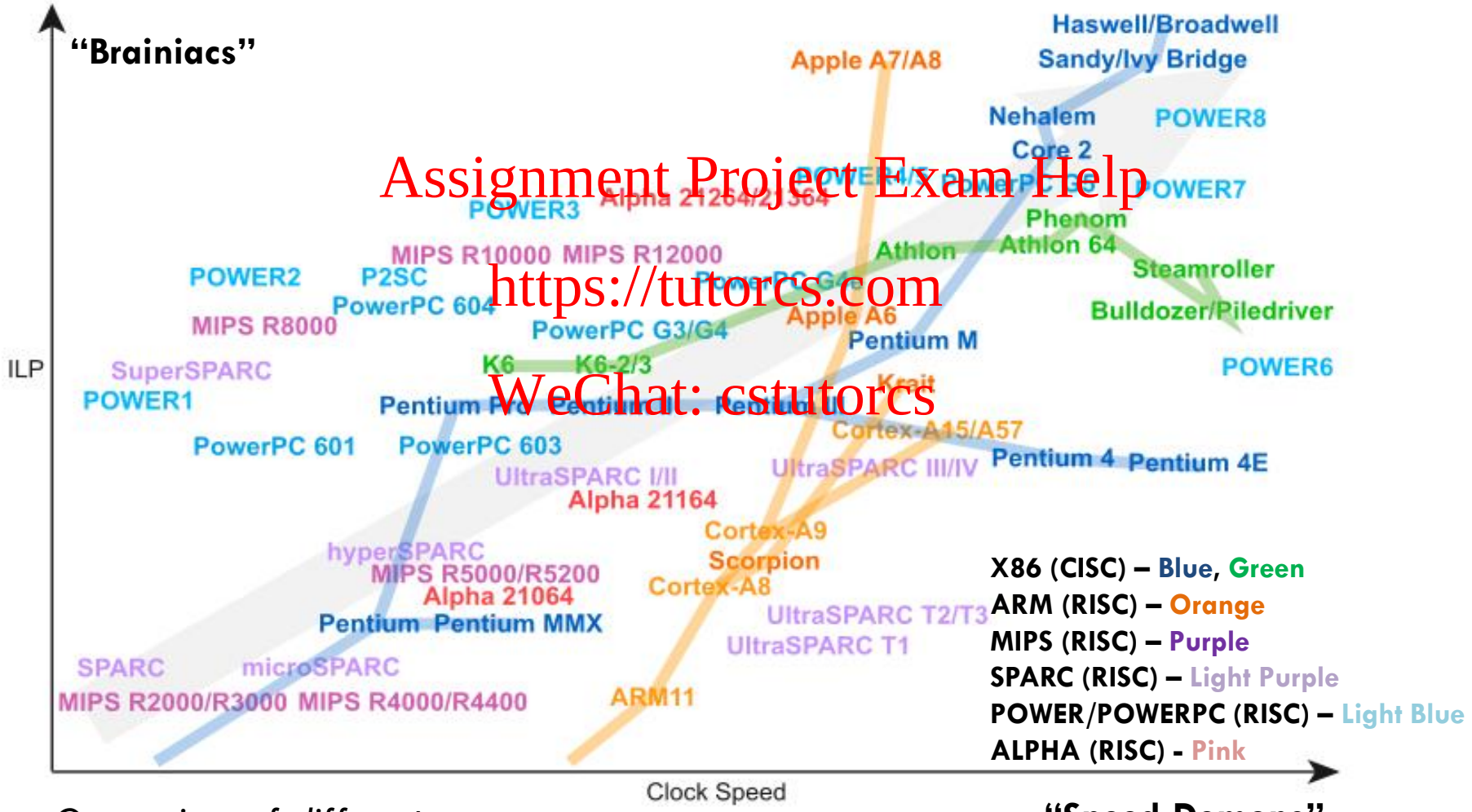
- Drawbacks over RISC & CISC
  - It is not always possible to completely fill all slots in a bundle, and empty slots are filled with NOPs
  - Still large code size
  - Poor compiler support can significantly impact the performance of EPIC code

Think-Pair-Share 2nd Exercise

Question: What is in your opinion the best Instruction Set Architecture (ISA) and why?

Answer: There is no good and bad ISA, but appropriate and not appropriate. The appropriate ISA depends on the target goals and on the target application

Comparison of different processors (Brainiacs vs Speed-Demons) (1)



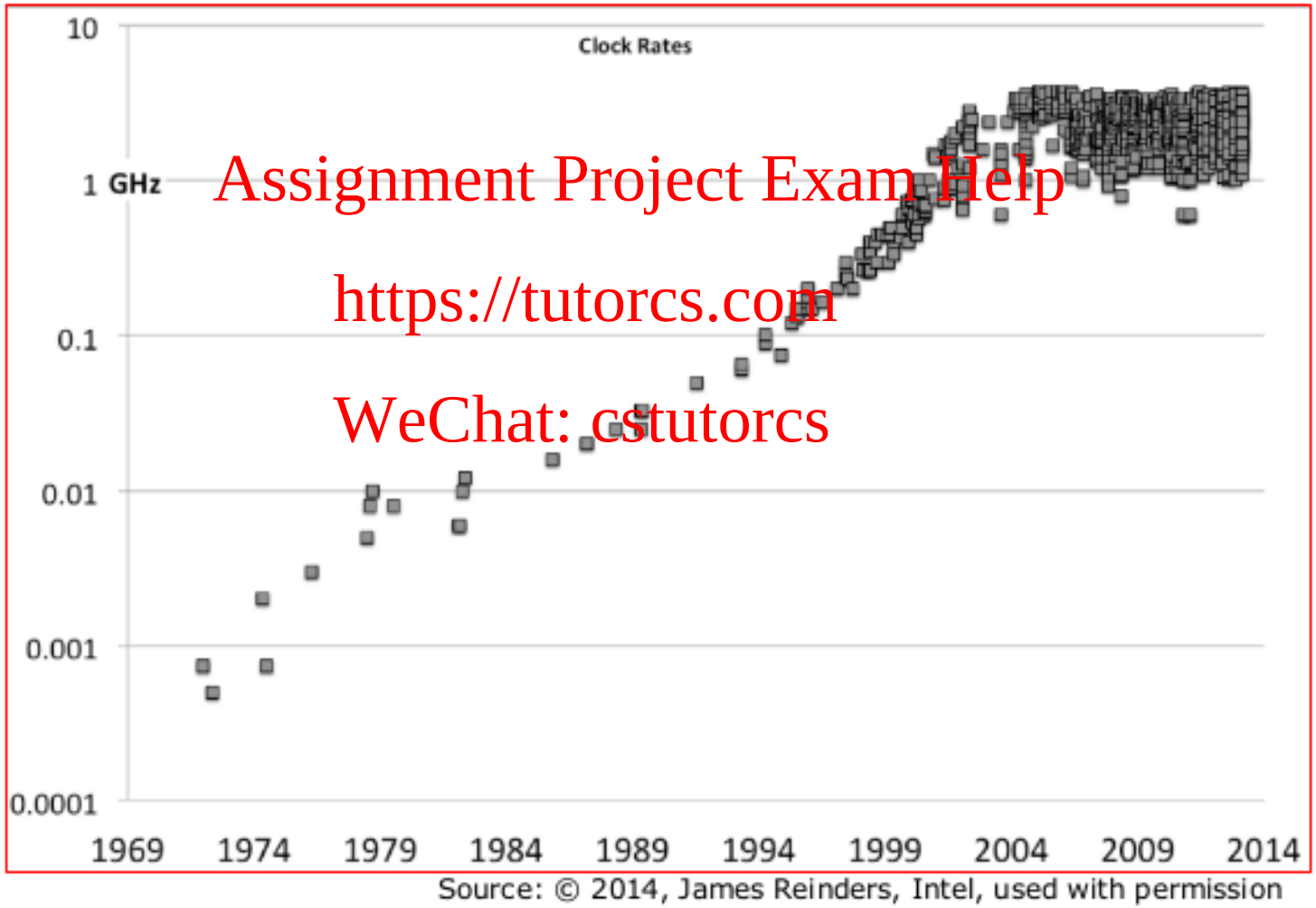
Comparison of different processors (http://www.lighterra.com/papers/modernmicroprocessors/)

Comparison of different processors (Brainiacs vs Speed-Demons) (2)

- "Brainiac designs"
  - Extra HW in order to achieve more Instruction Level Parallelism (ILP) out of the code
  - Millions of extra transistors
  - More design effort
  - Consume more power
- "Speed-Demons"
  - Run at higher clock speeds because they are simpler
  - Simple HW design
  - Less Chip area
  - Less power consumption

Which would you rather have: 4 powerful brainiac cores, or 8 simpler in-order cores? they use the same chip area

The CPU frequency has ceased to grow



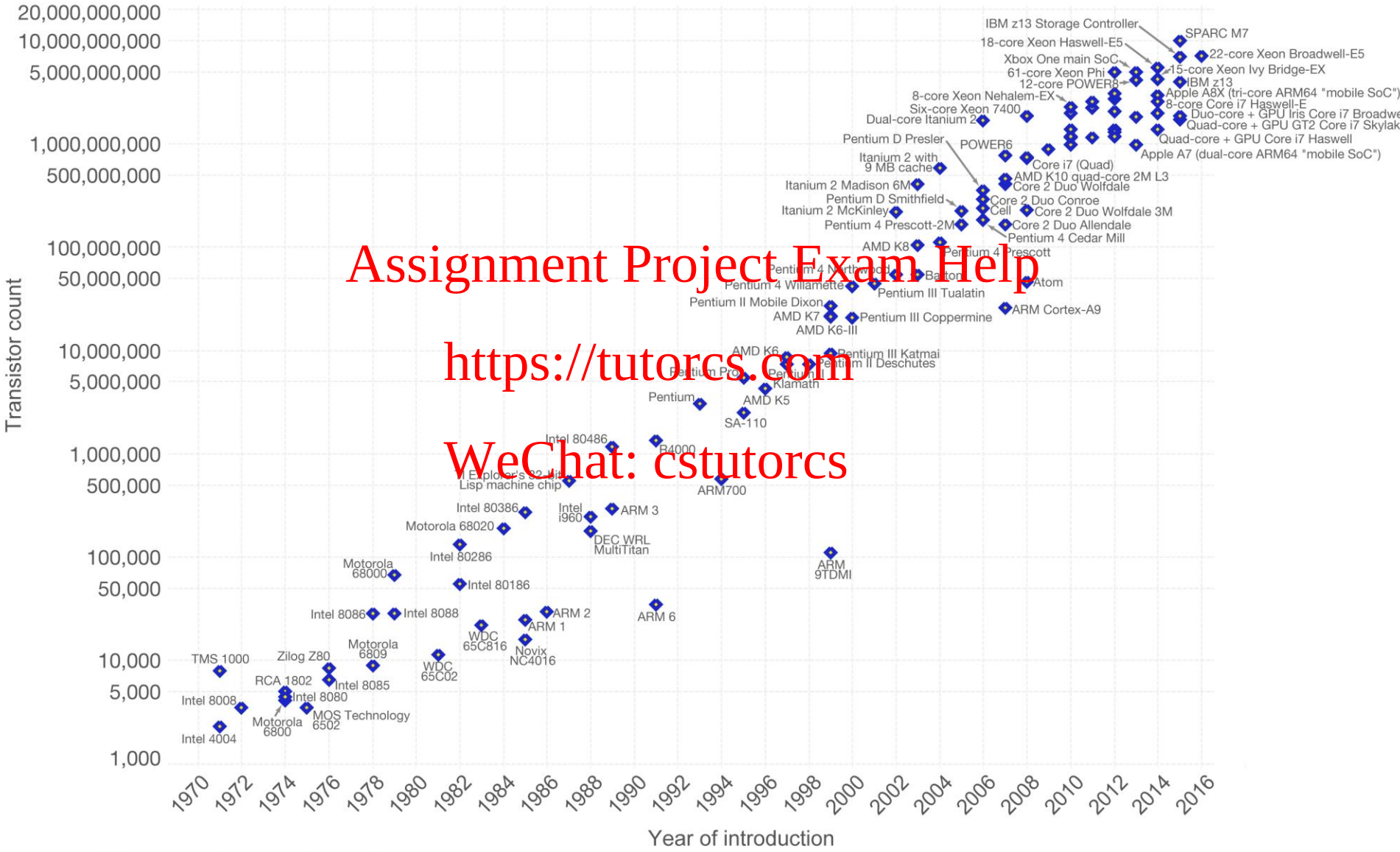
Moore's Law

- How small can we make transistors?
- How densely can we pack chips?
- No one can say for sure
- **Gordon Moore, co-founder of Intel, observed that the number of transistors per square inch on integrated circuits had doubled every year since the integrated circuit was invented**
- **Moore predicted that this trend would continue for the foreseeable future (Moore's law)**
- **In subsequent years, the pace slowed down a bit, but data density has doubled approximately every 18 months, and this is the current definition of Moore's Law.**
- **Most experts, including Moore himself, expect Moore's Law to hold true until 2020-2025**
- Using current technology, Moore's Law cannot hold forever
- There are **physical** and **financial limitations**
- Cost may be the ultimate constraint

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

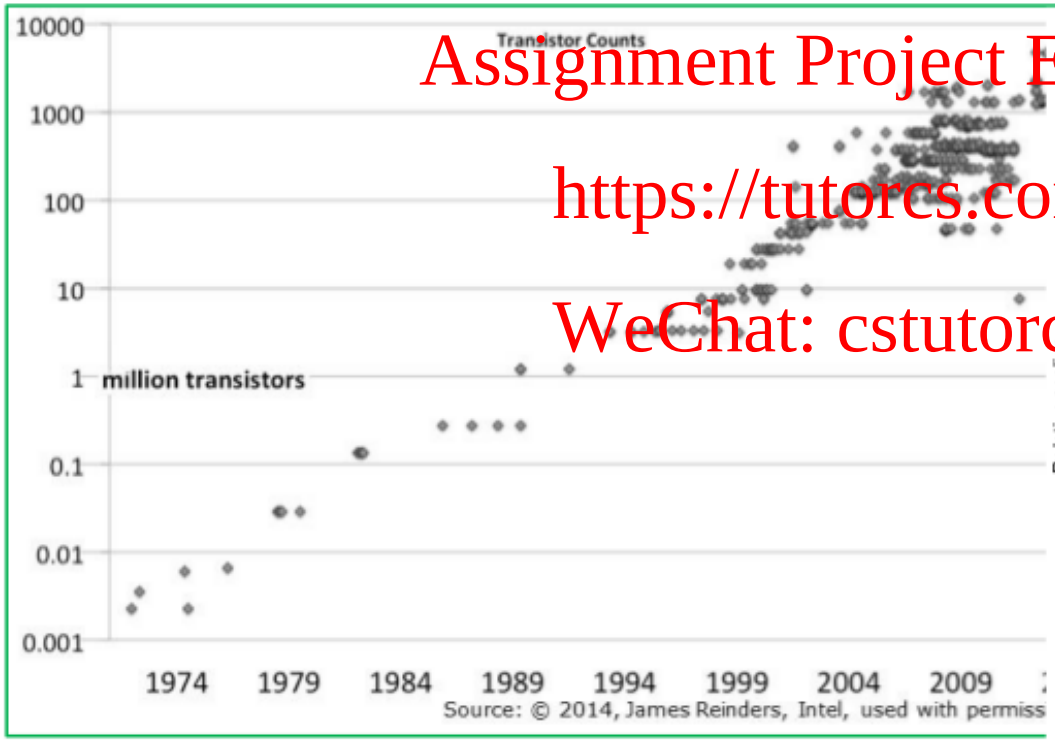
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

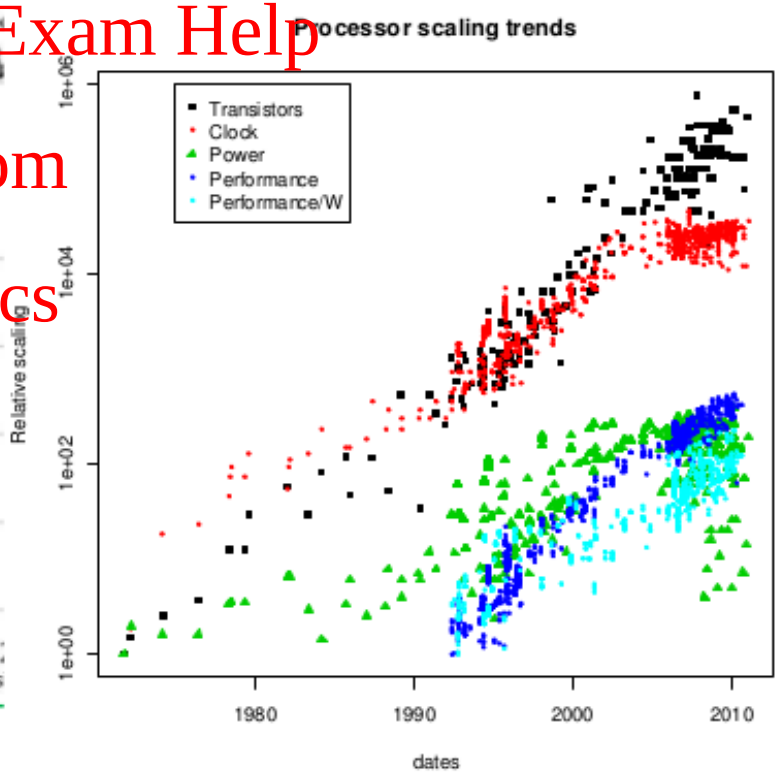
Moore's Law Is STILL Going Strong

Hardware performance potential **continues to grow**

"We think we can continue Moore's Law for at least another 10 years."

Intel Senior Fellow Mark Bohr, **2015**





Conclusions

- Computer architectures are complex and diverse
- There is a large number of different computer architecture classifications
- Classification helps us to select the most appropriate architecture
- Computer architectures are evolving year by year
- The computer architecture requirements are continually increasing
- There is no good or bad computer architecture. It depends on the
  - ✓ target goals, e.g., performance, flexibility
  - ✓ target application

Further Reading

Structured Computer Organization. Sixth Edition, Andrew S. Tanenbaum, Todd Austin, PEARSON, 2012, https://universalflowuniversity.com/Books/Computer%20Programming/Computers%2C%20Architecture%20and%20Design/Structured%20Computer%20Organization%206th%20Edition.pdf

Computer Organization & Architecture: Designing for Performance. William Stallings, Seventh Edition, 2006, available at https://inspirit.net.in/books/academic/Computer%20Organisation%20and%20Architecture%208e%20by%20William%20Stallings.pdf

Nicholas FitzRoy-Dale, The VLIW and EPIC processor architectures, available at https://www.cse.unsw.edu.au/~cs9244/06/seminars/02-nfd.pdf

Thank you