Chapter 3 Systems Approach - Modern Day Architectures & Performance Characterization

Contents of this chapter

- Design space of modern day computers
- Instruction pipelines
- Various instruction set architectures
- CISC & RISC processor systems
- Superscalar processors, Pipelining in superscalar processors, Hybrid processors

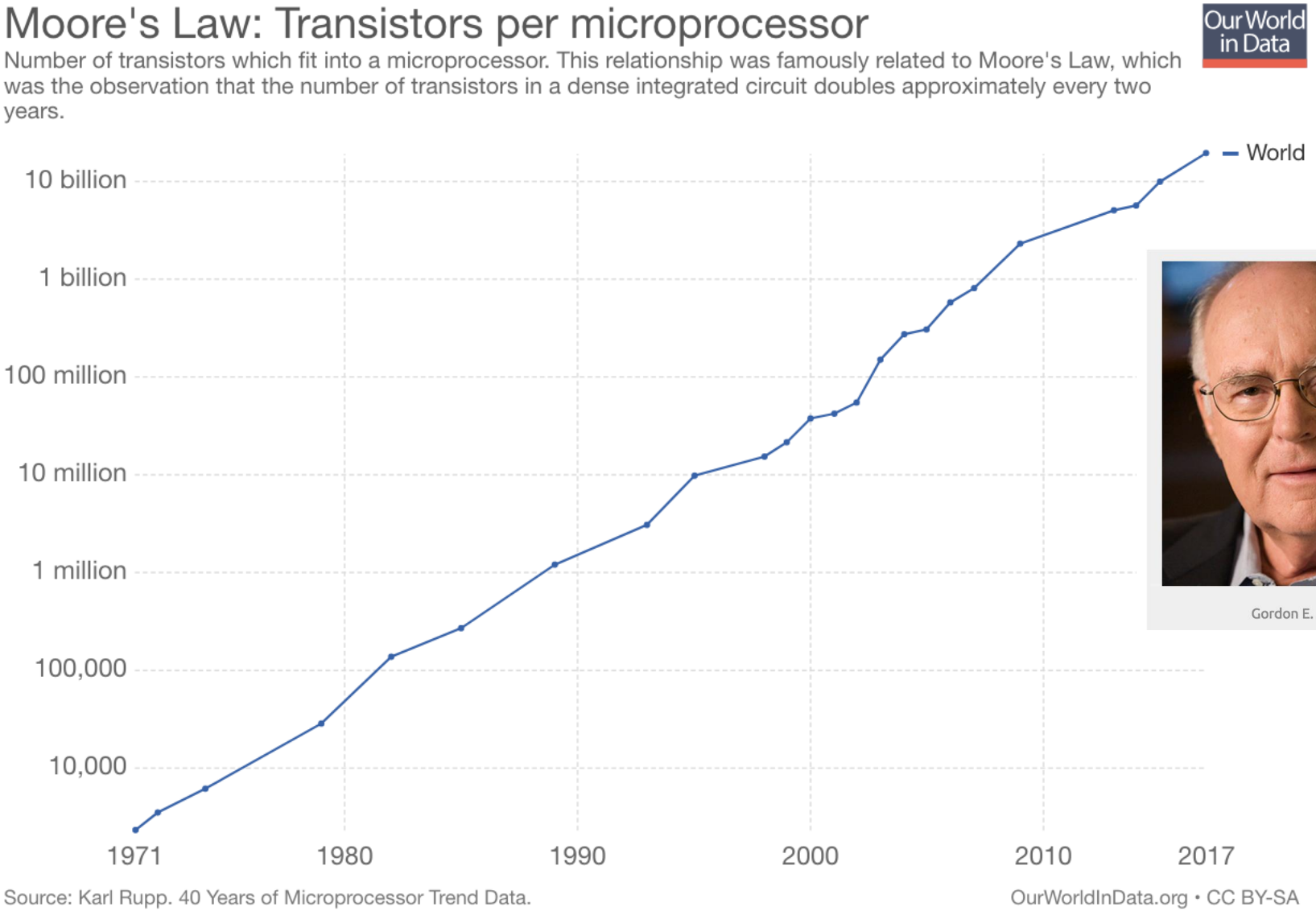
- VLIW Architecture
- Multi-threaded Processors

References *:*

1) Kai Hwang's Book on Advanced Computer Architecture
- Chapter 1 pages 3-46;

2) Kai Hwang et.al," Distributed and Cloud Computing: From Parallel Processing to the Internet of Things" ISBN-13: 978-0123858801 / ISBN-10: 0123858801, Publisher - Morgan Kaufmann book on Cloud Computing

Some performance indicators…



Gordon E. Moore

Moore's Law is a computing term which originated around 1970; the simplified version of ***this law states that processor speeds, or overall processing power for computers will double every two years***.

To break down the law even further, it accurate to say that the number of ***transistors*** *on an affordable CPU would double every two years*

If you were to look at processor speeds from the 1970's to 2009 and then again in 2010, one may think that the law has reached its limit or is nearing the limit. In the 1970's processor speeds ranged from 740 KHz to 8MHz;

From 2000 – 2009 there has ***not*** *really been much of a speed difference as the speeds range from 1.3 GHz to 2.8 GHz, which suggests that the speeds have barely doubled within a 10 year span. This is because we are looking at the speeds and not the number of transistors; in* ***2000*** *the number of transistors in the CPU numbered* ***37.5 million****, while in* ***2009*** *the number went up to an outstanding* ***904 million****; this is why it is more accurate to apply the law to the number of transistors than to speed.*

Some useful remarks ….

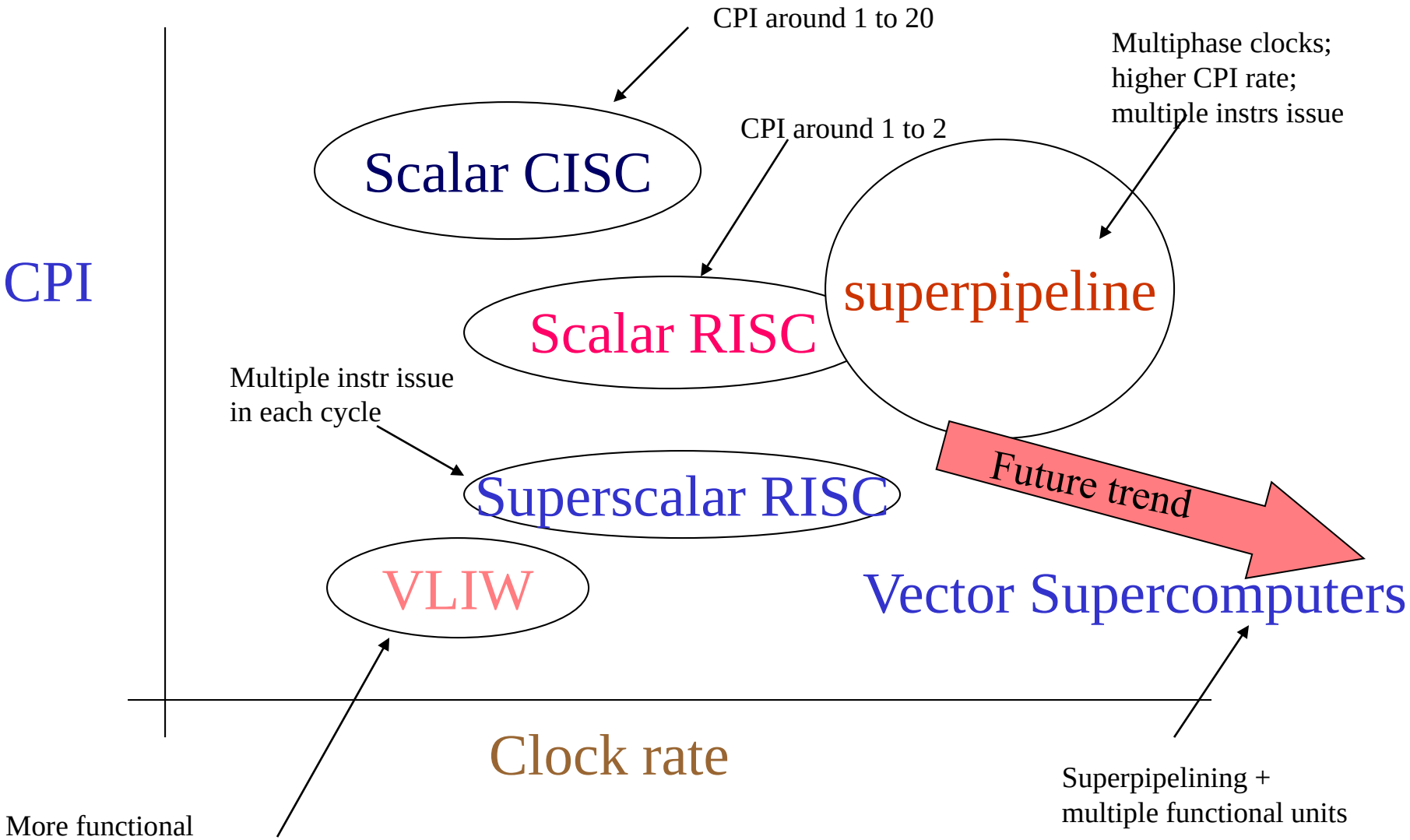
- Power grows with number of transistors and clock frequency
- Power grows with voltage; $P = CV^2f$
- Going from 12V to 1.1V reduced power consumption by 120x in 20 yrs
- Voltage went down to 0.7V in 2018, so only another 2.5x
- Power per chip peaking in designs:
- - Itanium II was 130W, Montecito 100W
- - Power is first-class design constraint
- Circuit-level power techniques quite far away still!

Design space of modern day computers

The design space implicitly considers the underlying microelectronics/packaging technologies.

A standard two dimensional plot to capture the technology growth of the current day computers is by representing clock rate vs cycles per instruction.

Cycles per instruction *versus* *Clock rate* reflects the technology advancement and the trend.



• Conventional computers : Classical examples - Intel i486, MC68040, VAX, IBM 390 belong to the family known as *Complex Instruction Set Computing (CISC) architecture*
(See the upper left corner)

• *Reduced Instruction Set Computing(RISC) architectures*

Earlier CPUs – ARC, ARM, Intel i860, SPARC, MIPS, IBM RS/6000, etc have faster clock rate and depends on implementation technology; With the use of advanced technology, the CPI of most RISC instructions has been reduced to one or two cycles

- *Superscalar processors*: This is a subclass of RISC family, which allows multiple instructions to be issued during each cycle. *This means that the effective CPI of superscalar processor should be lower than that of a generic RISC superscalar processor*.

- VLIW architecture: Uses sophisticated functional units to lower the CPI further. *Due to very long instruction word these processors are implemented using microprogrammed control*. Typically an instruction will be of 256 to 1024 bits in length.

- Superpipelined processors: These processors use *multiphase clocks* with an increased rate.

The CPI is rather high when multiple instructions are issued per clock cycle unless super-pipelining is adopted along with multi-instruction issue

However, *effective CPI of a processor used in a supercomputer should be very low owing to the use of multiple functional units*. The monetary cost is too high for the processors figuring in the lower left corner.

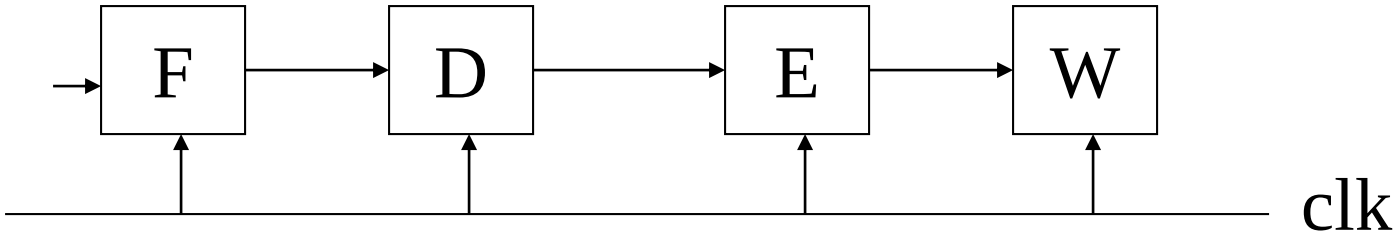
Instruction pipelines

Typical instruction execution is carried out in the following phases

- fetch -> decode -> execute -> write-back

Thus, the instructions are executed one after another, like an assembly line in a manufacturing environment.

Pipeline cycle: The time required for each phase to complete its operation, *assuming equal delay* in all the phases.



Some useful terminology: (*Refer to these definitions on Page 159*)

1. Instruction pipeline cycle : Clock period of the instruction pipeline (can be different from the Master clock)

2. Instruction issue latency : Delay (in terms of clock cycles) required between the issuing of two adjacent instructions.

3. Instruction issue rate : # of instructions issued per cycle, referred to as *degree of a superscalar processor*

4. Simple operation latency : Delay in carrying out simple (most of the times, this is the case!) operations (*add,load,store,branch,move, etc*). These latencies are measured in clock cycles

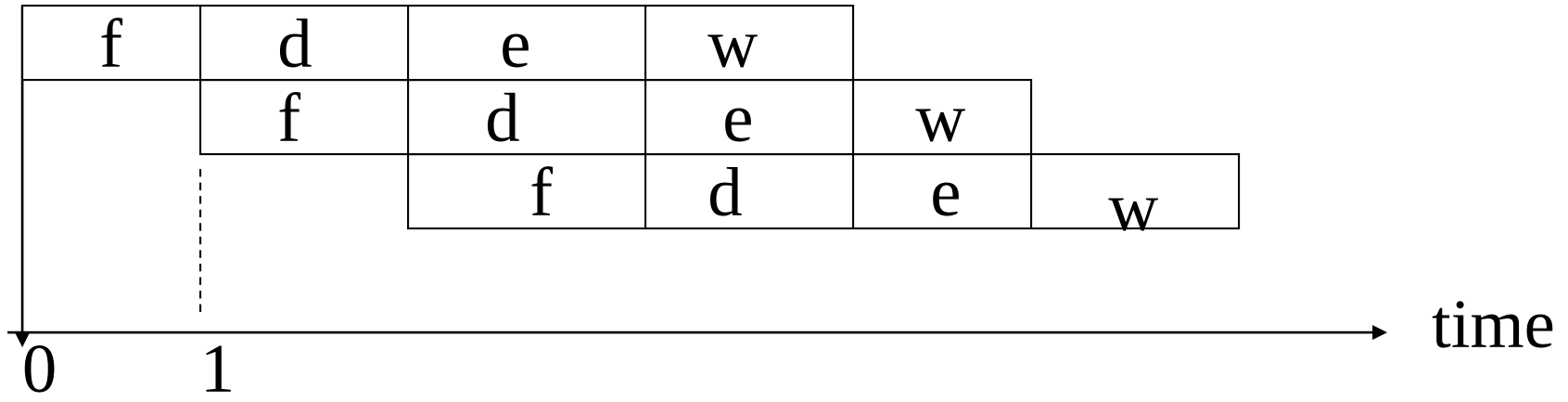
5. Resource conflicts : Demand for the same functional unit at the same time.

We will see all the design issues in the pipeline chapter

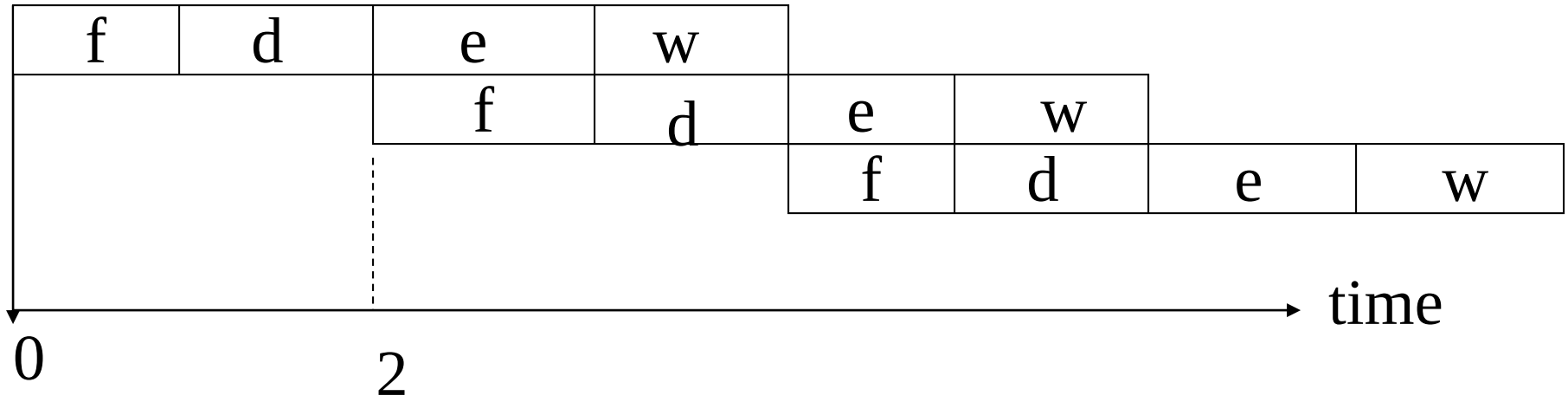
A *base scalar processor* is defined as a machine with

-- one instruction issued per cycle
-- one-cycle latency for a simple operation
-- one-cycle latency between instruction issues

The instruction pipeline can be fully utilized if successive instructions can enter continuously at a rate of one per cycle, as shown in the figure. Sometimes the pipeline can be underutilized, if instruction issue latency is two cycles per instruction. *See figures for other examples. Observe the effective CPI rating for each case*.



Base scalar processor (1 cycle per instruction)



Scenario showing 2 cycles per instruction

Remarks: *Some RISC processor pipelines distinguish between "MEM" and "WB";*
MEM – Memory operation; WB – Write Back
So, after EX stage, it could be either a MEM or a WB

WB corresponds to operations pertaining to writing into registers

Consider a set of instructions as follows.

I1: LD R1, 0(R2) ; load R1 from address 0 + R2 (mem to reg)
I2: ADD R1, R1, #1 ; R1 = R1 + 1
I3: ST 0(R2), R1 ; Store R1 at address 0 + R2 (reg to mem)

Coprocessors: The main CPU is essentially a *scalar processor*, which may consist of many functional units such as *ALU*, *a floating point accelerator*, etc.

The floating point unit can be on a separate processor, referred to as a *coprocessor*. This will execute the insts. issued by the CPU. In general, a coprocessor may execute:

-- floating point accelerator executing scalar data;
-- vector processor executing vector operands;
-- a digital signal processor;
-- a lisp processor executing AI programs

Note: A coprocessor cannot handle I/O operations

Instruction-Set(IS) Architectures

An instruction set of a computer specifies the primitive commands or machine instructions that a programmer can use in programming the machine.

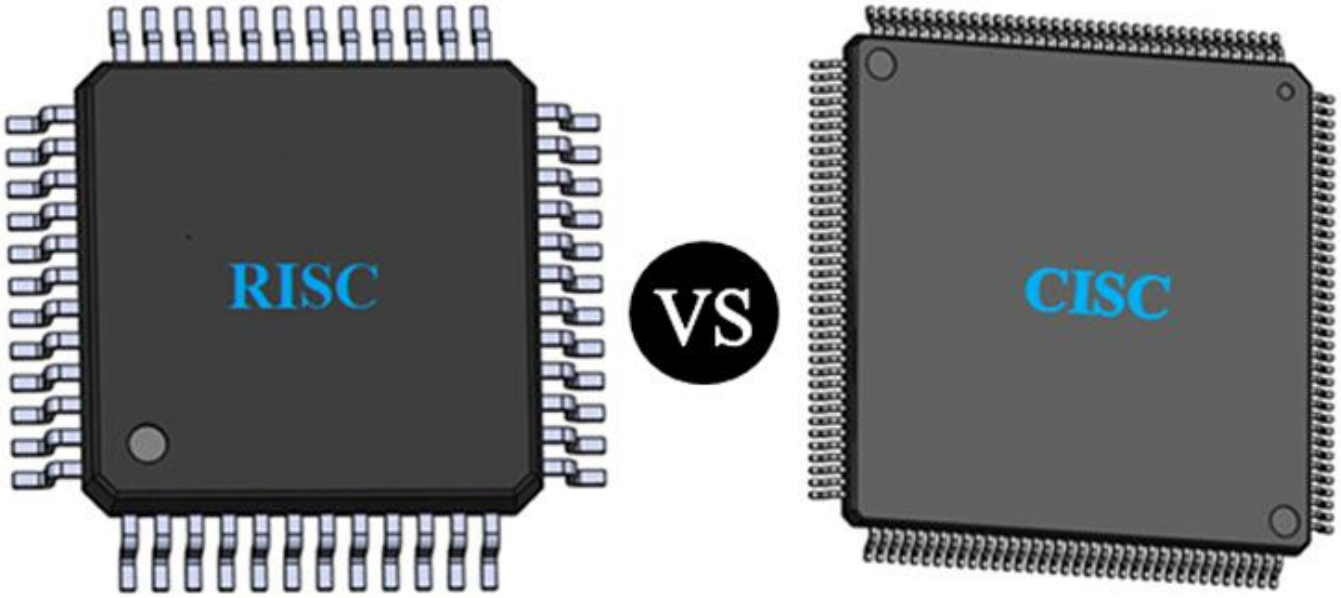
The complexity of an IS is attributed to the following:

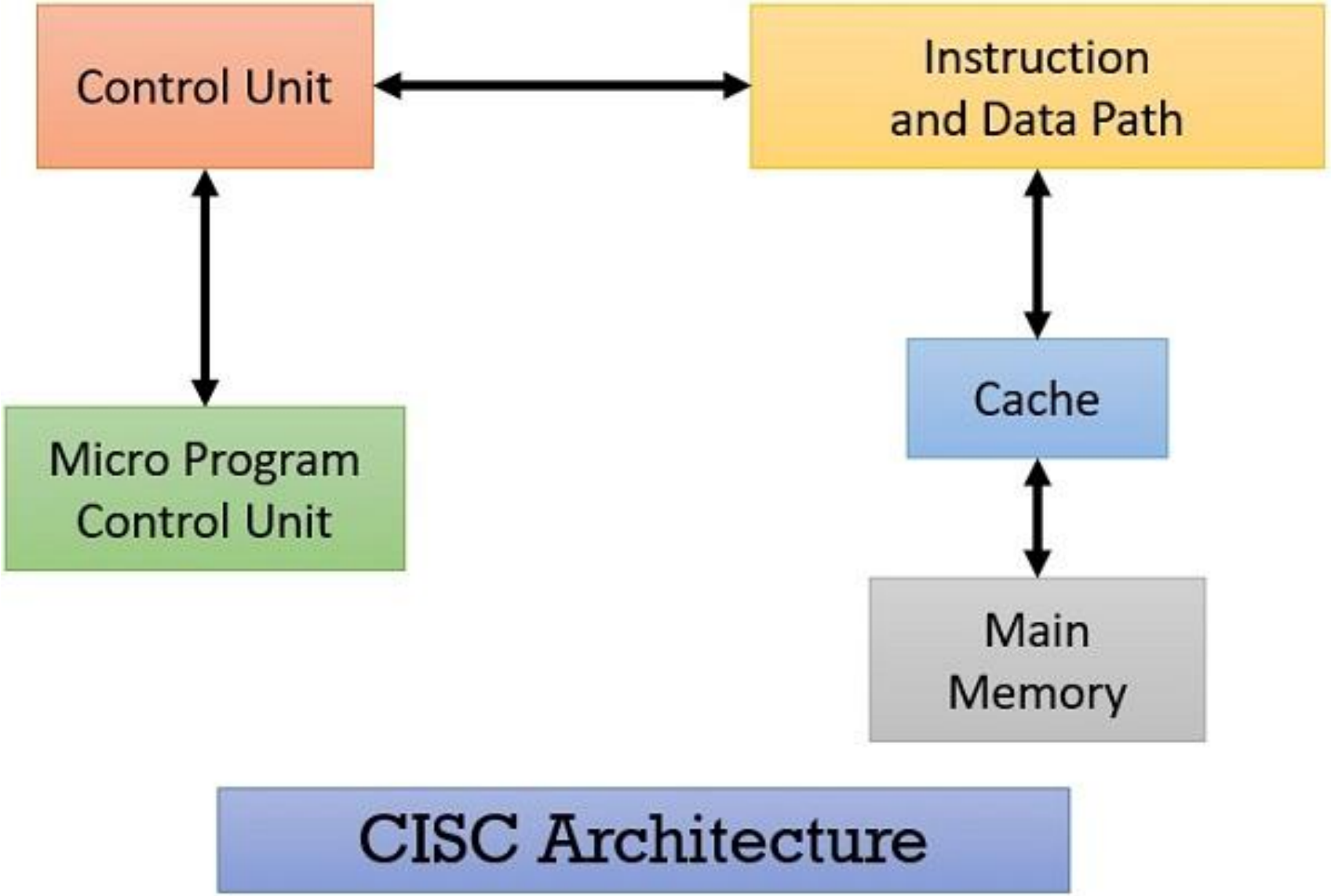
Instruction formats, data formats, addressing modes, general-purpose registers, opcode specs, and flow control mechanisms.

Based on the experience in the design of a processor, two different architectures have been proposed:

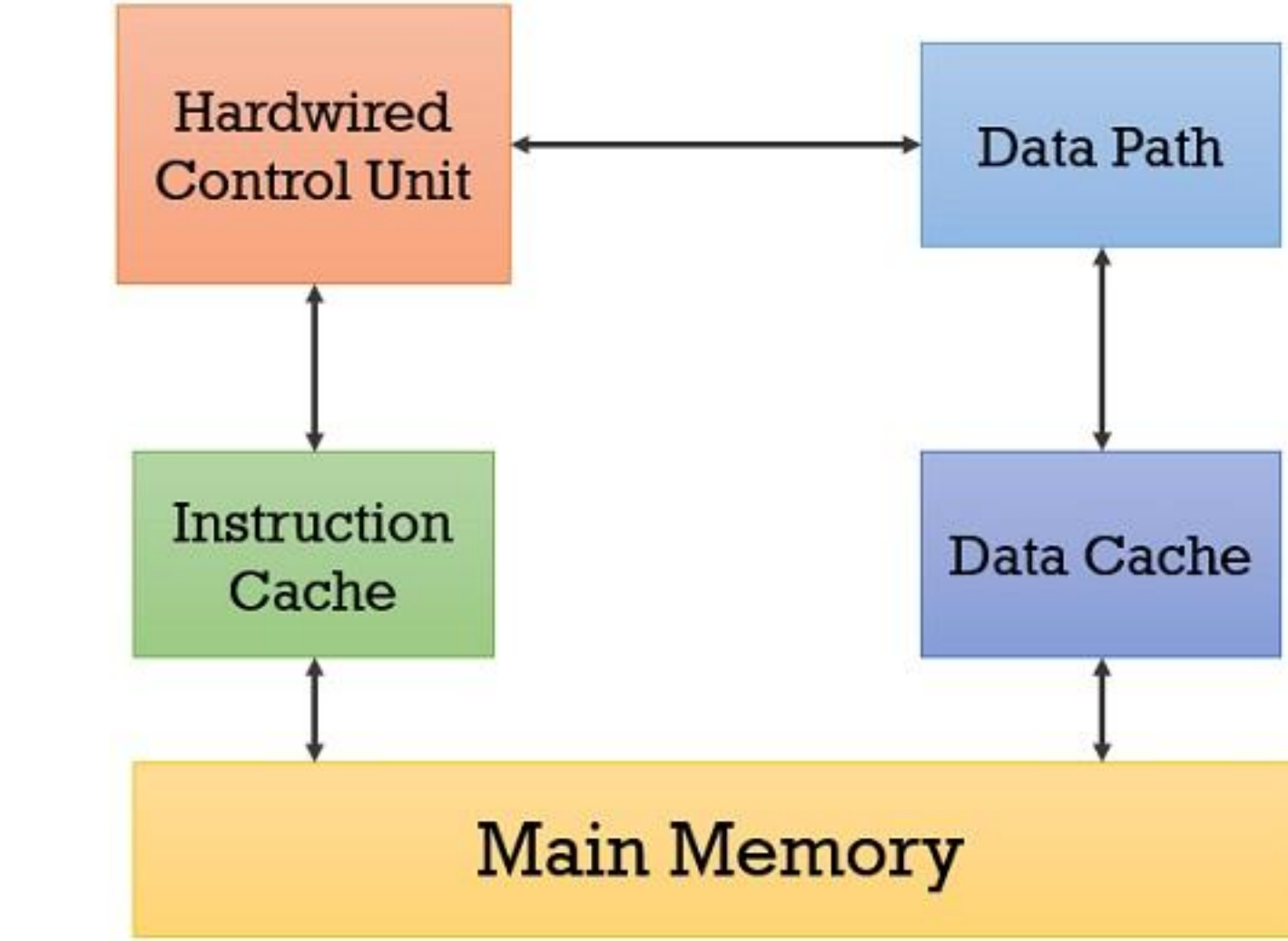
CISC (Complex instruction Set computing)

RISC (Reduced Instruction Set computing)





CISC Architecture



RISC Architecture

Architectural Distinctions

Characteristics	CISC	RISC
• IS/formats	Large sets with variable formats; 16-64bits/Inst 120-350 instructions typically	Small set with fixed formats(32-bits) Less than 100 instructions
• Addressing modes	12-24	3-5
• GPRs	8-24; data+inst in a single cache	32-192 split type cache

Characteristics	CISC	RISC
• Clock rate/CPI	x / 2 to 15	> k.x, k>1 / CPI < 1.5
• CPU control	micro-coded with control memory(ROM)	mostly hardwired without control memory

- Current day intel iCore series …. *wait!* ….and also recent past Pentium II/III/IV… series belongs to _____

Final remarks on RISC vs CISC

Based on the MIPS rate equation (*refer to Chapter 1)*, it seems that RISC will outperform CISC, *if the program length does not increase dramatically*. Expt. shows that converting from CISC program to an equivalent RISC program increases the code length by only 40%.

Some disadvantages of RISC architecture

1. It lacks some sophisticated instructions found in CISC processors.
2. It uses large register files, implying that the traffic between functional units is more and decoding is complicated.

What makes RISC actually special?

• The RISC gains its power by using the software support for less frequently used operations.

• The reliance on a good compiler is more severe in RISC than in CISC

• Instruction level parallelism is what is *always attempted to the maximum possible extent in this architecture*

(Fine-grain parallelism is what is always expected)

Overlapped Register Windows

A characteristic of some RISC processors is their use of overlapped register *windows to facilitate passing of parameters and to avoid saving and restoring register values*.

Each procedure/process call results in the allocation of a new window consisting of a set of registers from the register file for use by the new procedure/process. Windows for adjacent procedures have overlapping registers that are shared to provide the passing of parameters and results.

The register file consists of global registers and local registers.

Example : Let us say that the system has 74 registers on the whole.

Registers R0 through R9 : Global registers; These hold parameters shared by all the processors

Registers R10 to R73 (64 registers) are divided into say, 4 windows to accommodate 4 procedures. Each register window can have 10 registers (local) and a set of 6 registers common to adjacent windows

Local registers are used exclusively for local variables.

Common registers are used to exchange results and parameters between adjacent windows. The common overlapped registers permit parameters to be passed without actual movement of data.

Only one register window is activated at any time with a pointer indicating the current active window.

In general, the organization of the register windows will have the following relationships.

Number of global registers = G

Number of Local registers in each window = L

Number of registers that are common to 2 windows = C

Number of Windows = W

Thus, the number of registers available for each window is calculated as,

Window size = (L + C + G) (best case)

In our example: L = 10; C = 6; G = 10, W=4

The total number of registers needed in the processor (register file) is then given by,

Register file size = (L+C)W + G

Berkeley RISC I :

- 32 bit instruction format and a total of 31 instructions;
- Addressing modes: register, immediate operand, relative PC for branch instructions;
- 138 registers --- G = 10; W = 8 with 32 registers in each

Superscalar processors

The performance of CISC or RISC can be vastly improved using a superscalar or vector architecture.

Superscalar processor - multiple instruction pipelines
This means that *multiple instructions are issued per cycle and multiple results are generated per cycle*.

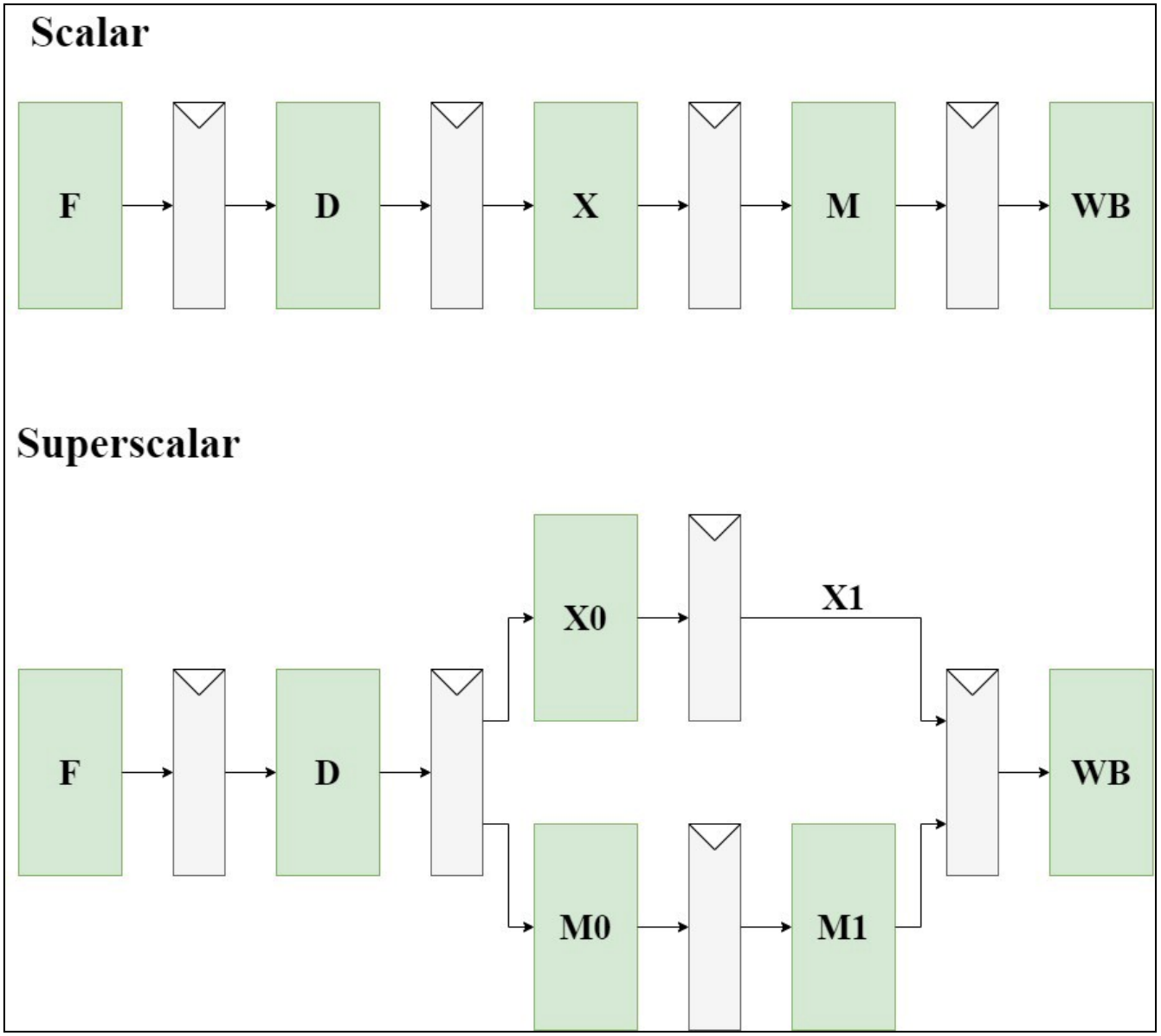
A vector processor executes vector instructions on arrays of data. This means that each instruction involves a string of repeated operations, which is most suitable for pipelining with one result per cycle.

Superscalar processors belong to the class of processors that are designed exclusively to exploit data-level parallelism combined with *instruction-level parallelism*

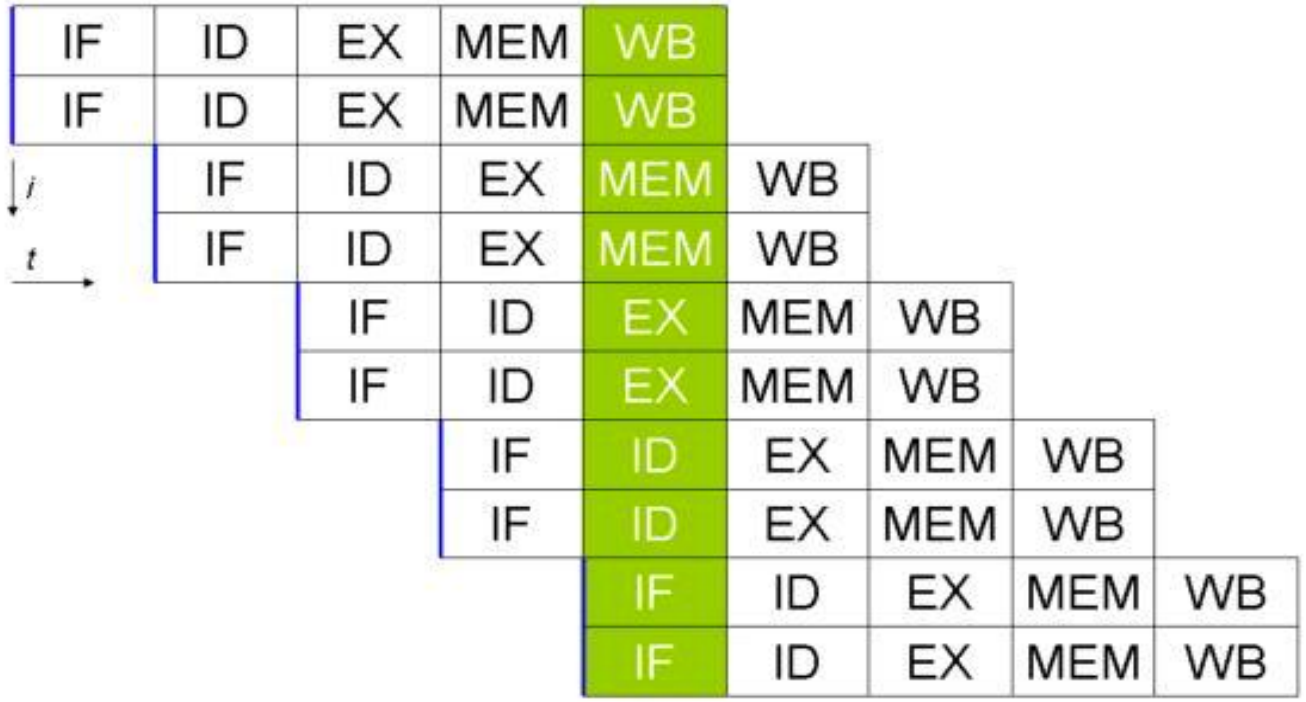
This is achieved by engaging multiple pipelines that can work concurrently;

In practice, it has been observed that the *instruction-issue degree* in a superscalar processor is limited to 2 to 5.

In general, a superscalar processor of degree m can issue m instructions per clock cycle. In order to fully utilize its power, m instructions must be executed in parallel.

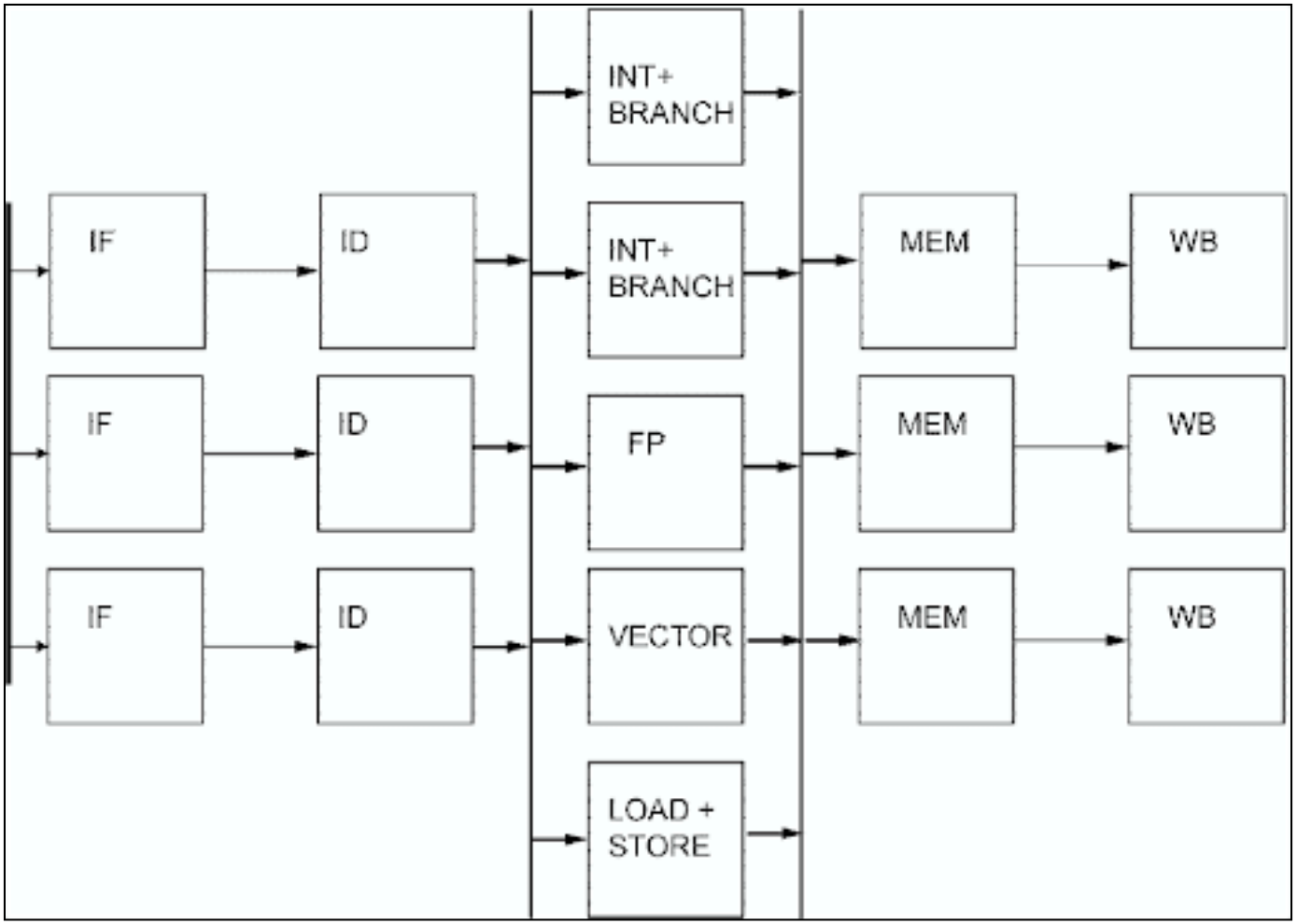


Two-issue Super Scalar CPU



Degree of super scaling m = 2

Superscalar processor with 3 pipelines
Degree of super scaling m = 3



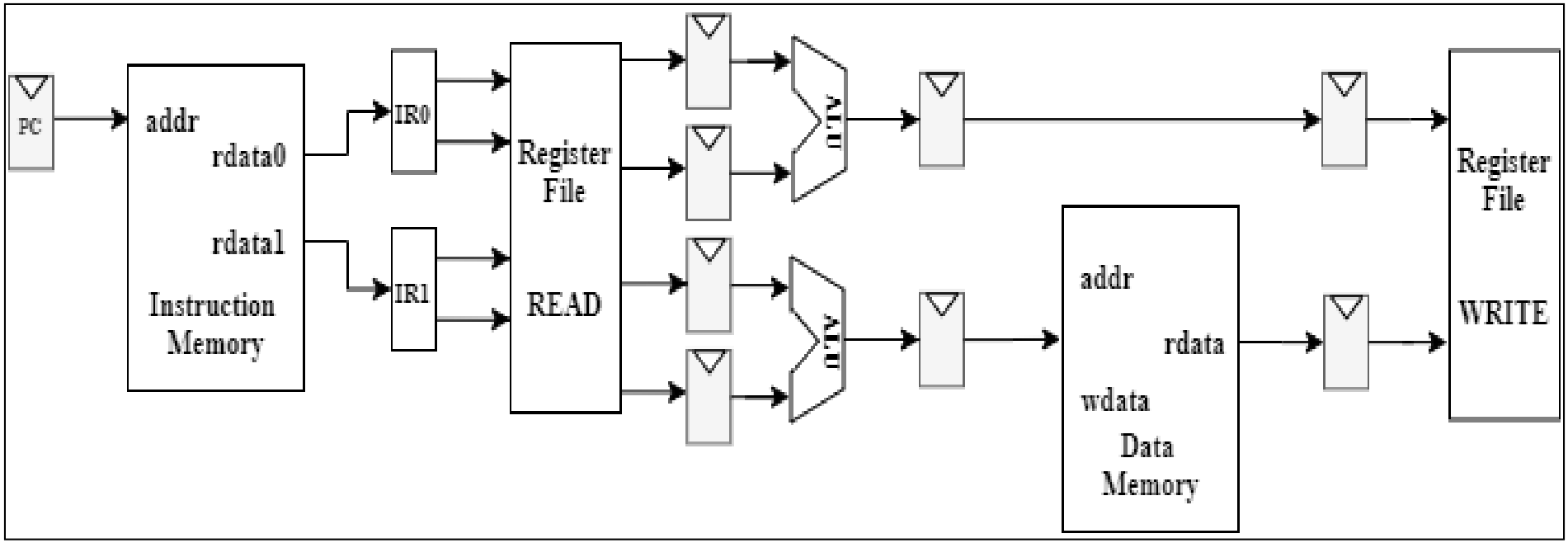
In-Order Superscalar Processor

- Fetches and runs multiple instructions from a thread of execution that preserves the compiler-generated order.

- In-Order processors are easier to implement, requiring less complex hardware (*at the cost of higher frequency of pipeline stalls*), which impacts negatively the system throughput.

Refer to the next slide for a 2-way Superscalar processor capable of performing 1 memory and 1 arithmetic operations concurrently.

A 2-way Superscalar Processor



Pipeline 1 (top) Arithmetic operation
Pipeline 2 (bottom)– Memory operation

Superscalar Detailed Example

We assume a superscalar pipeline is capable of:

- *fetching and decoding two instructions at a time*
- *having three separate functional units, i.e., two integer arithmetic and one floating-point arithmetic functional units*
- *having two instances of the write-back pipeline stage*

Given: *In-order issue & in-order completion constraints on a six-instruction code fragment:*

- *I1 requires two cycles to execute.*
- *I3 and I4 conflict for the same functional unit.*
- *I5 depends on the value produced by I4.*
- *I5 and I6 conflict for a functional unit.*

Instructions are fetched two at a time and passed to the decoder unit. Because instructions are fetched in pairs, the next two instructions must wait until the pair of decode pipeline stages has cleared;

Note: To guarantee in-order completion, when there is a conflict for a functional unit or when a functional unit requires more than one cycle to generate a result, the issuing of instructions temporarily stalls;

We need to see the performance due to the following 3 modes of operations:

- *In-order Issue & in-order completion*
- *In-order issue & out-of-order completion*
- *Out-of-order issue & out-of-order completion*

- *In-order Issue & in-order completion (8 units)*

Units \ Time	1	2	3	4	5	6	7	8
D	I1		I3		I5			
D	I2	I2	I4	I4	I6	I6		
E		I1	I1			I5		
E			I2				I6	
E				I3	I4			
W				I1	I3		I5	
W				I2		I4		I6

In this example, the elapsed time from decoding the first instruction to writing the last results is <u>eight</u> (8) cycles

- *In-order issue & out-of-order completion (7 units)*

Units \ Time	1	2	3	4	5	6	7	8
D	I1	I3		I5				
D	I2	I4	I4	I6	I6			
E		I1	I1		I5	I6		
E		I2						
E			I3	I4				
W			I2	I3		I5		
W				I1	I4		I6	

Here, as we can execute out of order, I2 starts at t=2 and also completes at t=3; Since out of order is allowed, I3 can start at t=3 (no constraint given for out of order completion);

- *Out-of-order issue & out-of-order completion (6 units)*

Units \ Time	1	2	3	4	5	6	7	8
D	I1	I3	I6	I5				
D	I2	I4	I4					
E		I1	I1					
E		I2		I6				
E			I3	I4	I5			
W			I2	I3	I6			
W				I1	I4	I5		

This is OOO issue and OOO Completion - For I1 and I2 no changes from earlier case. But I6 is issued first and then I5 because I5 is dependent on I4;

Pipeline Hazards

Main constraint – Concurrently processing the instructions by a Superscalar processor needs to guarantee *sequentiality of the operations* to ensure the correctness of the computation

Pipeline hazards - Situations that prevent the next instruction (from the instruction stream) to be executed in its designated clock cycle – resulting in pipeline stalling – *loss of pipeline efficiency!* [Data + Structure + Control hazards]

Data hazards - Read-After-Write (RAW), Write-After-Read (WAR) and Write-After-Write(WAW)

RAW hazard or Data Dependence:

RAW hazard occurs when instruction J tries to read data before instruction I writes it, in a seq:
I: R2 <- R1 + R3 J: R4 <- R2 + R3.

- When Read & Write operations are concurrently executed in a SS processor

Between instructions that reads a register and a subsequent instruction that writes the same register

Example: Assume that LD has a lower CPI than ADD

ADD R1, R2, R3 // R1 = R2+R3

LD R2, R1 // CPI is less than ADD

In this case, for ADD instruction, R2 may be loaded with an outdated value of R1, if LD has lower cpi than the ADD! **For ADD, R2 is Read After a Write happens in LD!**

WAR hazard or Anti-dependence:

- W*rite* and *Read* operations are performed on the same register, or memory address, processed concurrently by a SS processor,

WAR hazard occurs when instruction J tries to write data before instruction I reads it.
I: R2 <- R1 + R3 J: R3 <- R4 + R5.

Example: Assume that LD has a lower CPI than ADD

ADD R5, R3, R2 // Note - R3 is a src reg; **before R3 is read, LD writes to R3**

LD R3, R7 // CPI is less than ADD

WAW hazard or Output Dependence:

- Concurrent write operations with the same destination register in a SS processor

Example: (second must be processed after the first for correct results)
ADD R1, R4, R6
ADD R1, R3, R5

How to take care of pipeline hazards?

- By *halting* a pipeline for the required number of cycles (refer to RAW example and realize that we need to delay the 2nd pipeline by 1 clock cycle)
- Other techniques – *Register forwarding, Scoreboarding* [*Chapter 4 (if time permits)]*

Structural hazard

- Refers to a situation where *different instructions require access to the same resource simultaneously*;

- The pipeline stalls the next instruction until the resource is available, delaying the execution of the instruction stream;

Control hazard

- Refers to a situation when the pipeline misses the branch prediction and consequently reads instructions that should not be executed;

- The whole pipeline data has to be discarded and the correct instructions are to be re-fetched

Q: Where and what are the hazards in the following execution sequences?

LD:	*IF*	*D*	*EX*	*MB*	*WB*				
Add:		*IF*	*D*	*EX*	*MB*	*WB*			
Add:			*IF*	*D*	*EX*	*MB*	*WB*		
ST:				*IF*	*D*	*EX*	*MB*	*WB*	

Original Addr seq; New location

0x50	*ADD:*	*IF*	*D*	*EX*	*MB*	*WB*			
0x54	*JMP 0x10:*		*IF*	*D*	*EX*	*MB*	*WB*		
0x58	*Add:*			*IF*	*D*	*EX*	*MB*	*WB*	
0x10	*ST:*				*IF*	*D*	*EX*	*MB*	*WB*

Solution approaches:

Q1 – LD & ST clash for memory resource in the 4th cycle (MB,IF); Delay IF of ST by ~~1 Cycle~~ - Structural Hazard

Q2 - Control Hazard; The program execution jumps to a non-sequential address. This operation requires that the sequential instructions that are already fetched and being processed in the pipeline be killed and the instructions at the new address fetched. The killed instructions inject NOP operations in the pipeline, cleaning the stages and avoiding bypassed values to interfere with the execution of the instruction at the new address. So, we need to show that Add at 0x58 will have NOP from stage D to the last – In fact, all instructions will be flushed from the original seq and new instructions are fetched from 0x10 as shown;

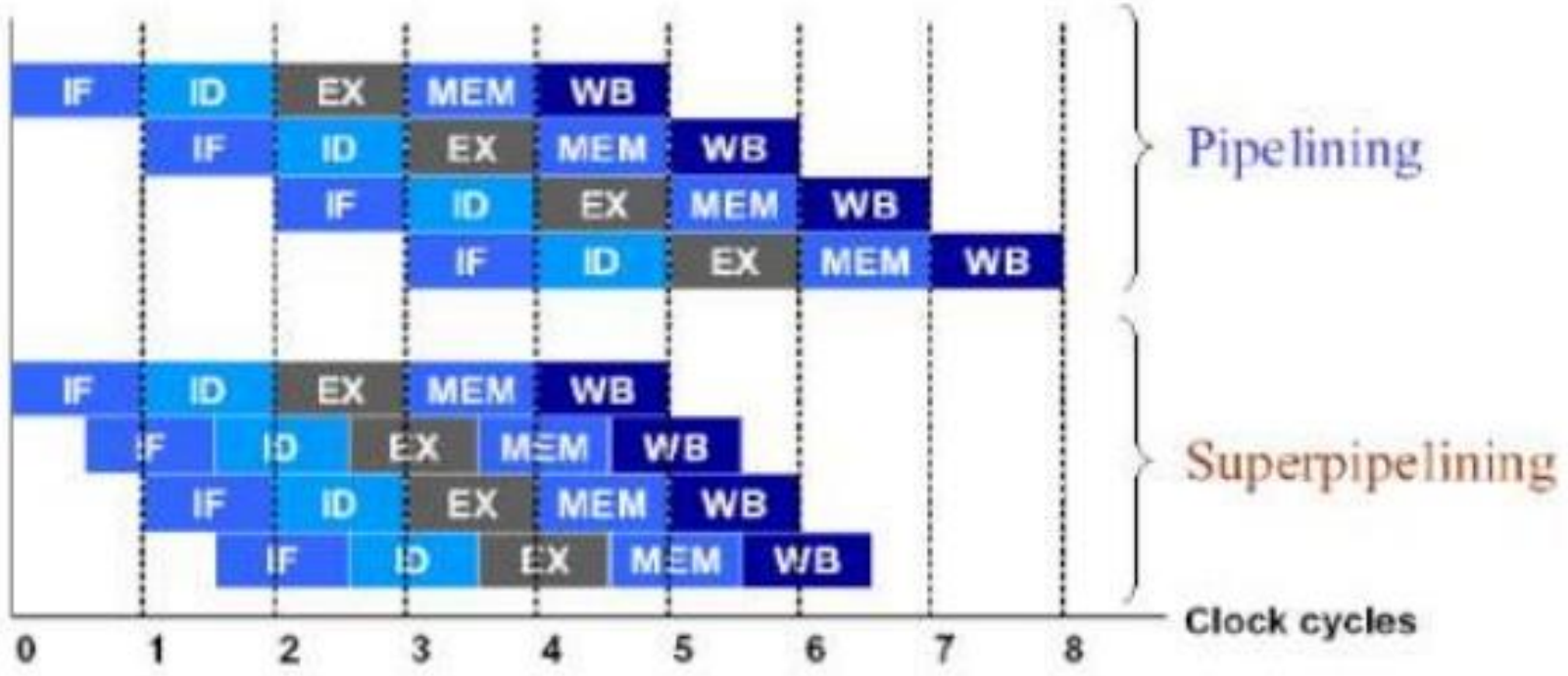
Superpipeline Processors

Extension of pipelining idea here! In normal pipelining, each of the stages takes the same time as the external machine clock. But, not all pipeline stages need the same amount of time. For example, *instruction decoding (especially in a RISC machine) is faster than the other stages such as fetches and stores.*

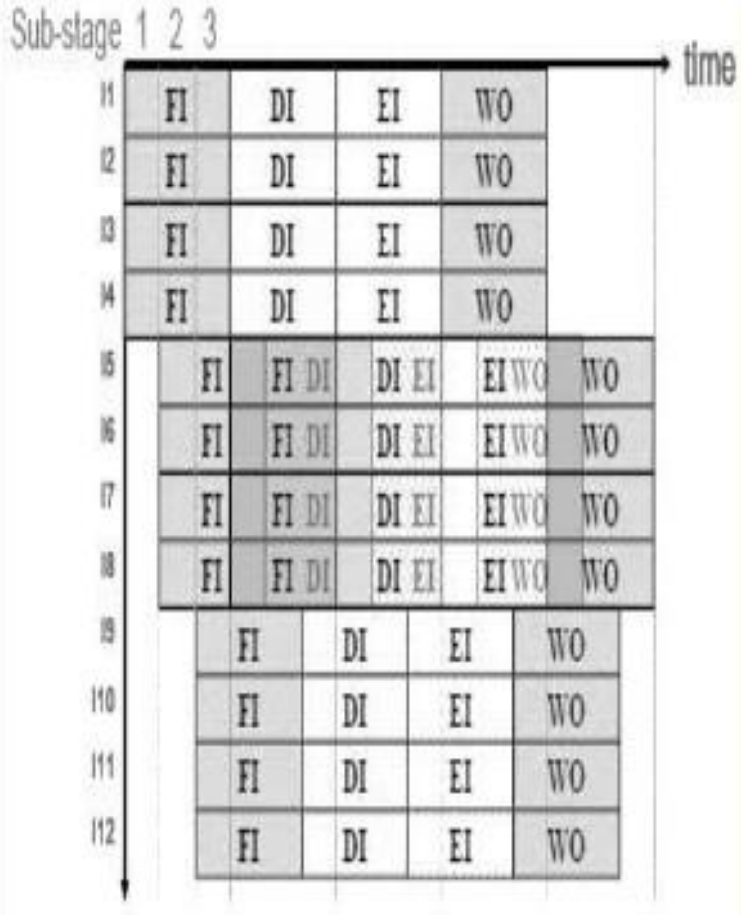
A superpipelined architecture exploits this by having an internal clock which is faster (typically double) than the external clock. Thus, in the same external clock cycle, we can overlap two subtasks of two different instructions.

Super pipeline Performance

- The performance is shown below in the figure:



Superpipelined Superscalar



Superpipeline of degree 3 and superscalar of degree 4:

- 12 times speed-up over the base machine.
- 48 times speedup over sequential execution.

Some examples:

- Pentium Pro(P6): 3-degree superscalar, 12-stage "superpipeline".
- PowerPC 620: 4-degree superscalar, 4/6-stage pipeline.

16

VLIW Architecture

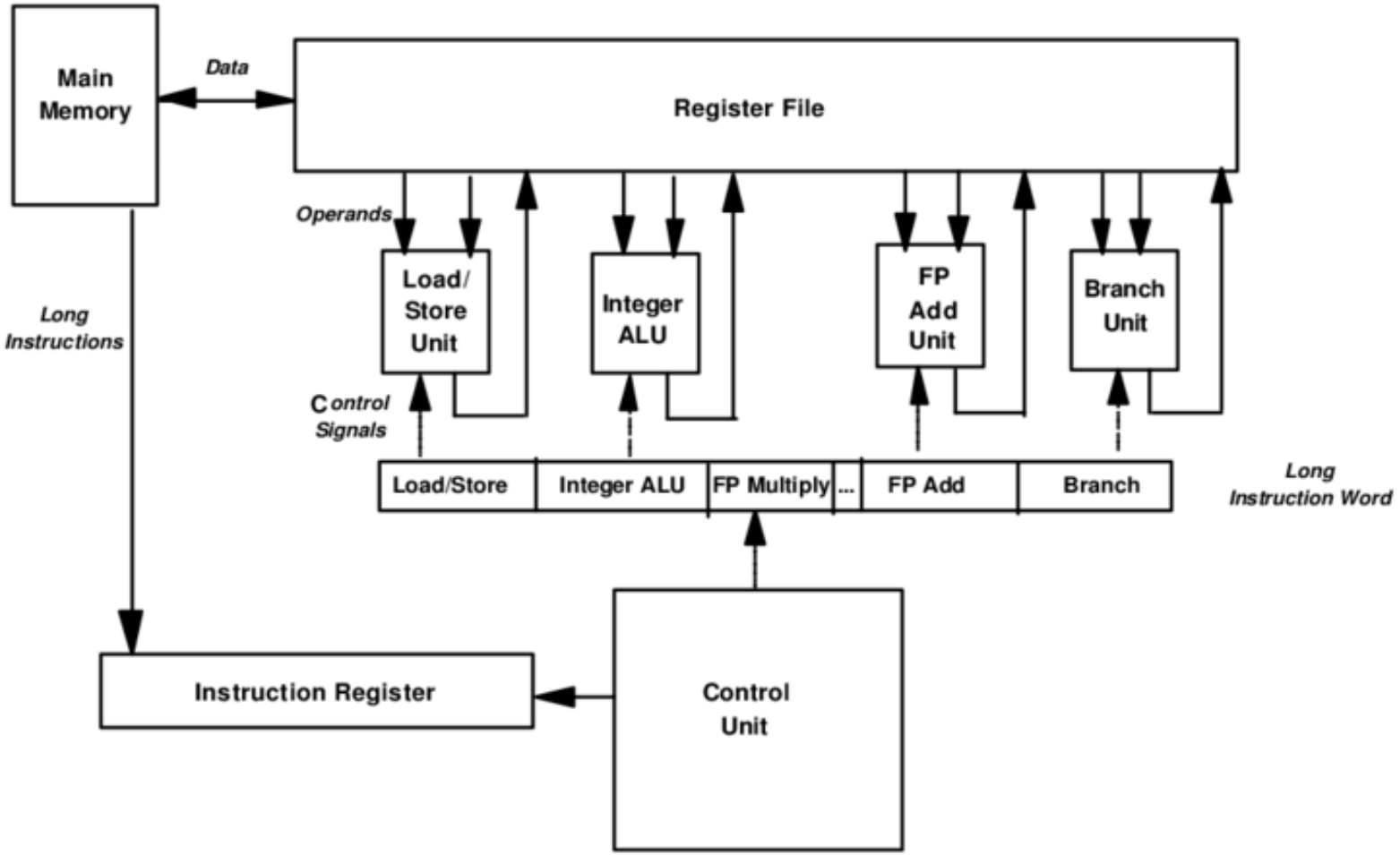
- Generalization of two concepts: *horizontal micro-coding* and *superscalar processing*

A VLIW machine has instruction words hundreds of bits in length. (*See Fig. next slide*)

Multiple functional units are used concurrently and all these units share a common register file. *Those operations that are required to be executed simultaneously by the functional units are synchronized in a VLIW instruction, say, 256 or 1024 bits per inst. word.*

Pipelining in VLIW machines

Here, each instruction specifies multiple operations.



Instruction level Parallelism

Instruction Level Parallelism (ILP) is a measure of how many operations in a program that can be performed simultaneously

The overlap among instructions is called ILP

Example:
Op1 e = a + b
Op2 f = c +d
Op3 m = e*f

Op1 and Op2 are independent; If we assume that each operation takes 1 unit of time, then ILP for this example is 3/2

VLIW attempts to exploit ILP available

Example (ILP in VLIW)

Computation: $y = a_1*x_1 + a_2*x_2 + a_3*x_3$

Seq Processor:

Load a1
Load x1
Load a2
Load x2
MUL z1,a1,x1
MUL z2,a2,x2
ADD y,z1,z2
Load a3
Load x3
MUL z1,a2,x3
ADD y,y,z2

VLIW Processor (2 load/store units, 1 MUL, 1ADD)

Load a1 Load x1	Load a2 Load x2 MUL z1,a1,x1	Load a3 Load x3 MUL z2,a2,x2
MUL z3,a3,x3 ADD y,z1,z2		
ADD y,y,z3		

Seq Proc: 11 Cycles
VLIW: 5 Cycles

Superscalar versus VLIW

It is important to note that VLIW processors *behave* much like superscalar machines, however, there are three distinct differences.

1. Decoding is found to be easier than superscalar instructions

2. The code density of the SS processors is *better when the available instruction-level parallelism is less than that exploitable by VLIW machine*. This is due to the fact that a VLIW machine has bits for non-executable operations while SS machines issue only executable instructions.

3. SS machines can be object-code compatible with a large family of non-parallel machines. But, a VLIW machine exploiting different amounts of parallelism would require different instruction sets.

Remarks: Inst. Parallelism and data migration in a VLIW machine are completely specified at the compile time and decision during *run time are not* needed. Thus, *run time scheduling and synchronization* are completely eliminated. We can consider a VLIW machine as a SS machine in which all independent and unrelated operations are synchronously compacted in advance. The CPI for VLIW *can be* less than SS machines.

Applications

- VLIW suitable for DSP applications
- Processing media like Compression/Decompression of Image and Speech data

Examples of VLIW

VLIW Mini supercomputers:

Multiflow TRACE 7/300, 14/300, 28/300
Multiflow TRACE/500
IBM Yorktown VLIW Computer

Single Chip VLIW Processors

Intel iWarp, Philip's LIFE Chips

DSP Processors (Ti TMS320C6x)

Multithreaded Processors

We define a "thread" as a short sequence of instructions schedulable as a unit by a processor.

A process in contrast normally consists of a long sequence of instructions which is scheduled by the operating system to be executed on a processor. Consider the following loop:

```
for I = 1 to 10
a(I) = b(I) + c(I);
x(I) = y(I) *z(I); end for
```

This loop can be unrolled and a thread can be created for each value of I. We will then have 10 threads which can potentially be executed in parallel provided sufficient resources are available.

Three types of MPs exist.

1. Blocked MPs

2. Interleaved MPs

3. Simultaneous MPs

All the above types have a basic pipelined execution unit and in effect, try to make best use of pipeline processing.

1. Blocked MPs

In this MP a program is broken into many threads which are independent of one another.

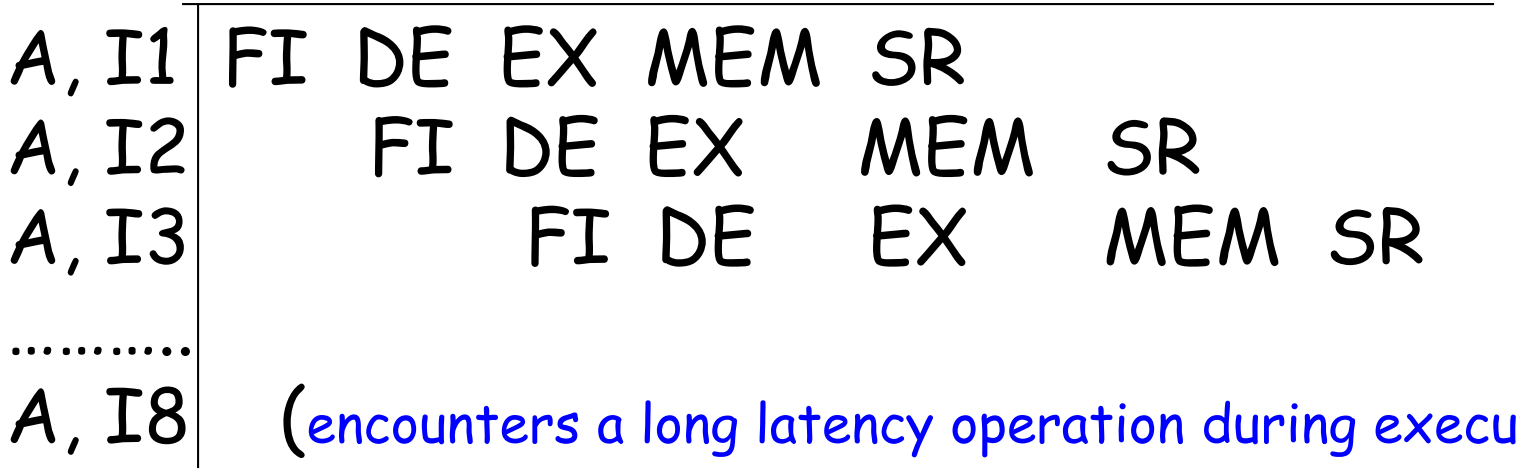
Each thread has an independent stack space, but shares a common global address space.

Consider the following example to see how this blocked MP works.

Assume that there are 4 threads A, B, C, and D which are ready to run and can be scheduled for execution.

Let thread A be scheduled first. Let instructions I1, I2, I3,..I8 be scheduled in that order.

The progress of I8, for example, in the pipeline may be delayed due to data dependency, cache misses, etc. If the delay is just 1 or 2 cycles, we can tolerate it. However, if the delay is



several cycles, then the processor will suspend A and will "switch" to B.

Before switching to thread "B", the "status" of A should be saved so that it can be resumed when the chance for A comes.

The status of A consists of PC, Program Status word, instruction registers and processor registers. It is, of course, not appropriate to store it in MM because it will take several cycles to store and retrieve them.

So, the best solution is to have a set of status registers reserved for each thread and simply switch from one set to another.

The above solution is feasible if the number of threads is not too many.

Major questions we should ask here are,

1. How many threads should be supported?

2. What is the processor efficiency?

In order to answer these questions, we must identify the parameters which affect the performance of the processor. These are as follows.

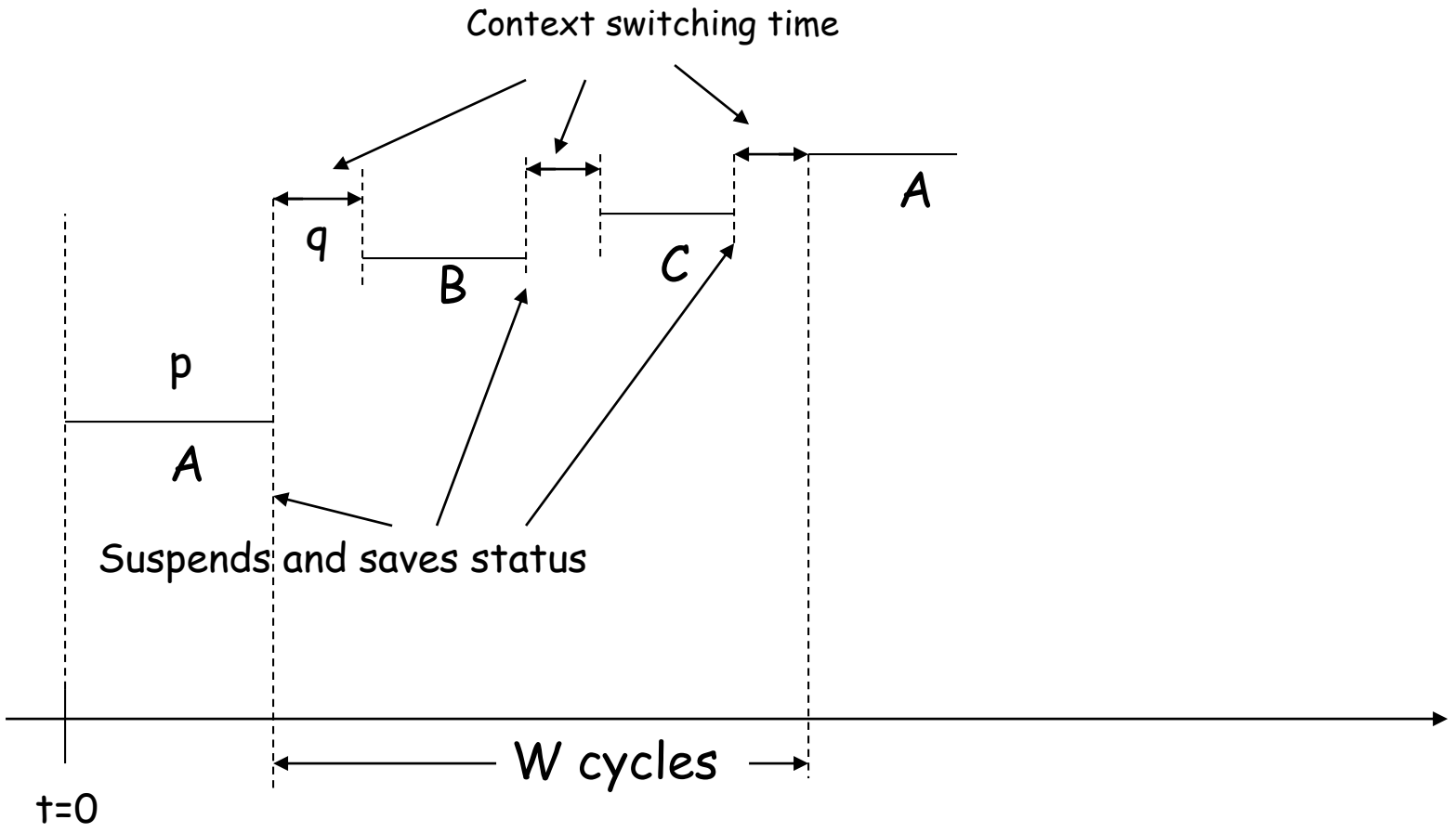
1. Average number of instructions which a thread executes before it suspends - say p

2. The delay when a processor suspends a thread and switches to another one - let this be q cycles

3. Average waiting time of a thread before it gets the resource it needs to resume execution called latency - let it be w cycles

We will count all these in number of processor cycles.

To determine the number of threads n to reduce the latency effects, we use the formula, as per the following figure.

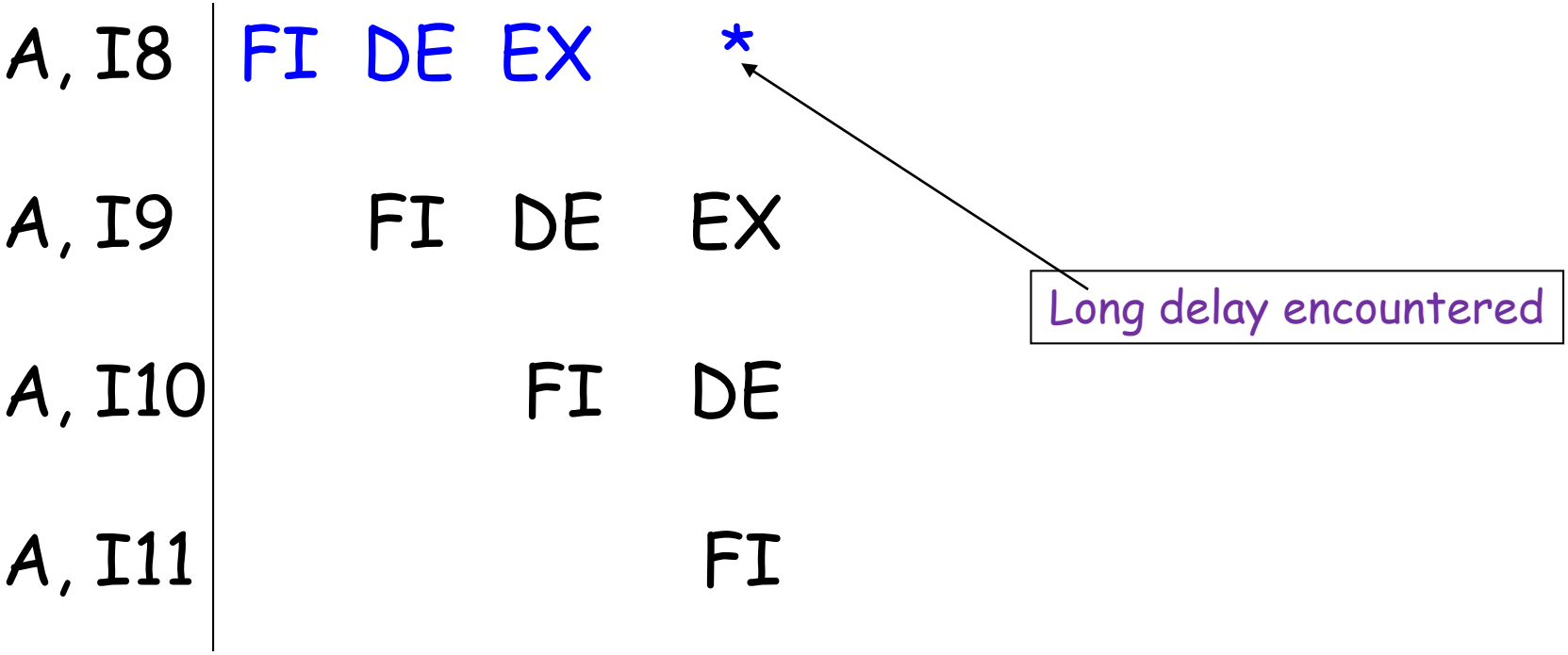


n q + (n-1)p <= w This implies that the number of threads that can be supported is

n <= (p+w)/(p+q)

Example:

Suppose p = 7 and w = 15. If the number of pipeline stages is five and a cache miss occurs in MEM cycle, three more instructions would have been issued and the previous instruction would not have been completed.



The best policy is to let the previous instruction complete its work and *abandon* the next three instructions.

In the above figure, if the delay occurs when I8

(cont'd) is being executed in its MEM cycle, we should allow I7 to complete and abandon I9, I10, and I11. *Why to abandon them?*

The reason we abandon the succeeding instructions is that they may depend on the suspended instruction. Thus, we should wait for at least 1 cycle in this example(SR operation) to complete I7 before switching to another thread. Therefore, q = 1 in this example.

Note: In longer pipelines, q will be large.

In the worst case, if instruction fetching is delayed, the delay can be equal to the depth of the pipeline.

Thus, in our example,

$$n = (7+15)/(7+1) \sim 3$$

Observe that larger p and smaller w reduces the number of threads to be supported.

Efficiency: Roughly this is calculated as

$$p/(p+q)$$

The efficiency is given by ~88% in our example. This poor efficiency is due to the fact that the average non-blocked length of a thread, in this example, is small relative to the context switching time.

Blocked multithreading has poor performance because the instructions in the pipeline when the thread is suspended are abandoned and this equals the length of the pipeline in the worst case.

It is too expensive to invoke another thread while allowing to continue this delayed thread.

2. Interleaved MPs

Interleaved multithreading solves the problem with the blocked multithreading scheme. Here, every instruction is issued from a different thread.

Thus, instruction I1 from A is issued first, I1 from thread B is issued next and so on. Suppose let us say that the number of threads equal to the length of the pipeline, say 5 stages (5 threads).

After 5 processor cycles (assuming one cycle per

stage), I2 of *A* is issued and the I1 of *A* would have been completed. Thus, I2 will not be delayed due to any data or control dependency.

However, if I1 of *A* is delayed due to data not being in the cache, then it is better to delay *A* and take an instruction from another "ready" thread.

In this architecture, each processor has its own "private" set of registers including the status registers. When a thread is suspended due to long latency operation, all its registers (including the GPSs in a register file), are "frozen".

When the next thread is invoked, the processor switches context to the new thread's registers. In other words, each thread must have a private set of registers so that this switching can be done in just one cycle.

In this case, estimating the number of threads that can be supported is tricky as we need to know the probability of long latency operations.

Assuming we know that, then the minimum # of threads that can be run can be estimated as (d +w.q),
where, w = num of cycles needed for a blocked thread to resume, d - depth of the pipeline, q - probability of long latency operations

Observe that longer the depth of the pipeline, larger are the number of threads required.

First commercial machine (HEP - heterogeneous element processor) supported 128 threads.

Tera - multithreaded processor supports 128 threads, having pipeline depth of 16 and has 32KB on-processor memory for registers (32 per thread) and a 1K branch target registers. Here there is a provision to perform three operations can be performed simultaneously with every instruction issue.

3. Simultaneous MPs

In this case, many threads which can be potentially carried out in parallel, are kept ready. Two or more instructions are issued simultaneously per cycle, just as in superscalar architectures. In the succeeding cycles, instructions from other threads are scheduled.

Assumptions in this proposal

- enough cache is present to store data apart from the set of registers
- enough functional units are available

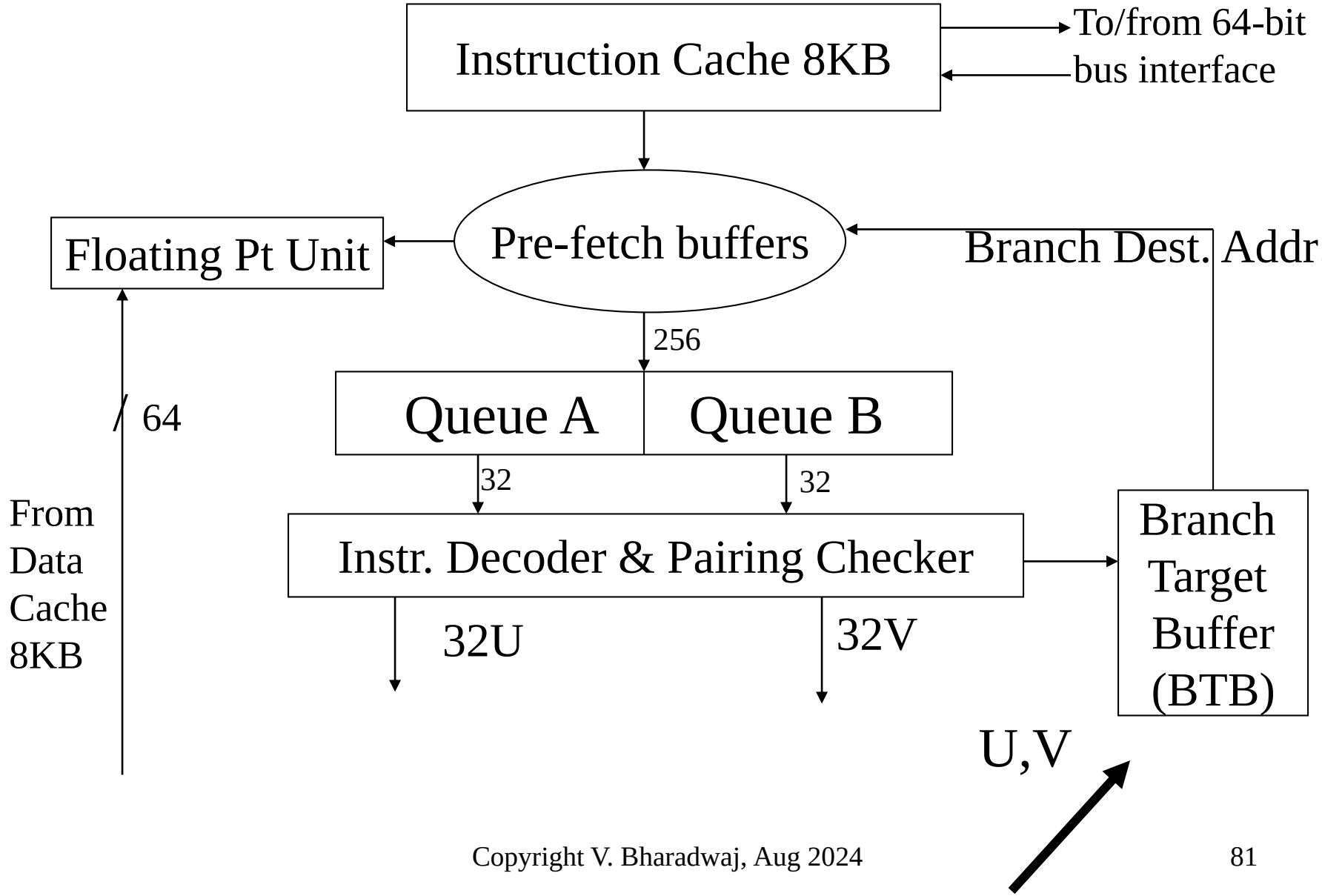
```
A, I1     FI DE MEM EX SR
B, I1     FI DE MEM EX SR
C,I1         FI DE MEM EX SR
D, I1        FI DE MEM EX SR
E,I1            FI DE MEM EX SR
F, I1           FI DE MEM EX SR
```

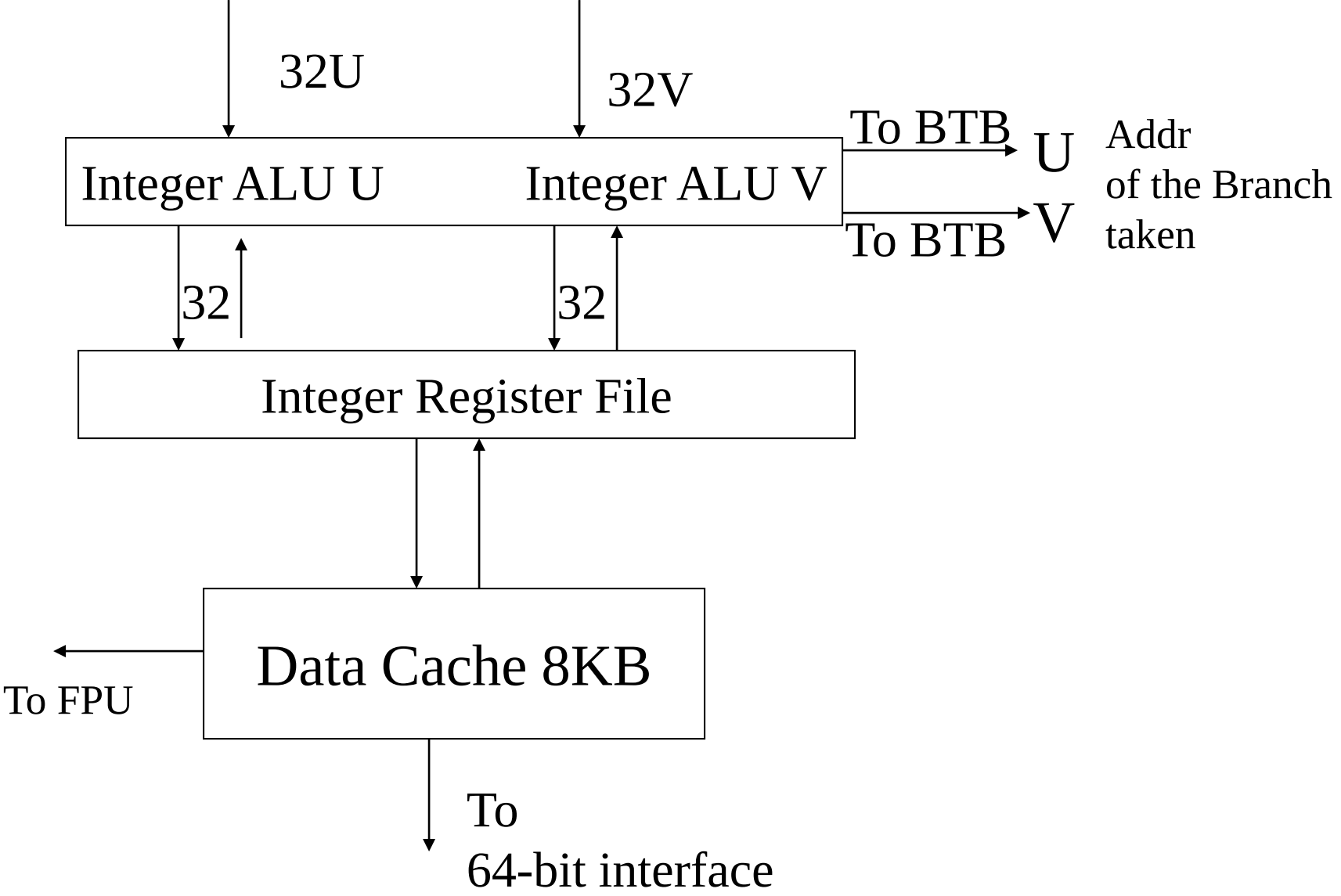
Motivation is to <u>combine multithreading and superscalar features</u> – Happening in GPU architectures now!

Earlier versions: IBM – Power 5 Processor; MemoryLogix- for mobile devices; Sun Microsystems – 4-SMT Processor CMP; Intel's Xeon;

ANNEXURE

Case study on *Intel Pentium CPUs*





Pentium uses 5 stage pipeline structure – Prefetch, 2 Decoding stages, Execute, write back;

Instructions are variable in length and stored in Prefetch Buffers;

In Dec-1 stage, processor decodes the instruction field and finds the opcode and addressing info; Checks which instructions can be paired for simultaneous execution and participates in branch address prediction;

In Dec-2 stage, addresses for memory reference are found;

In execute stage, data cache fetch or ALU or FPU operation may be carried out; *note that 2 operations can be carried out concurrently, if possible*;

In write back stage, registers and flags are updated based on the results of execution;

Pentium has two ALUs called U and V and hence 2 Instructions can be executed simultaneously.

However, some constraints exist to ensure potential Conflicts. *What are these*?

Two successive instructions I1 and I2 can be despatched in parallel to the U and V units provided the following 4 conditions are satisfied:

1. I1 and I2 are simple instructions (Simple instruction means an instruction can be carried out in 1 Clk Cycle.

2. I1 and I2 are not flow-dependent or O/P dependent; That is, dest. register of I1 is not the source of I2 and vice-versa;

3. Only I1 may contain an instruction prefix.
(Instruction prefix is of 0 to 4bytes long – address size, operand size, segment register the instruction must use, use of memory)

4. Neither I1 nor I2 contains (both) a displacement and an Immediate operand

---

BTB stores information about recently used Branch instructions; When an instruction is fetched BTB is checked. If the instruction address is already there, it is a "*taken branch instruction*" and the history bits are checked to see if a jump is predicted.

If YES, the branch target address is used to fetch the next instruction.
If NOT, it is updated.

Instructions in pre-fetch buffer are fed into *one of the two Queues A and B*.

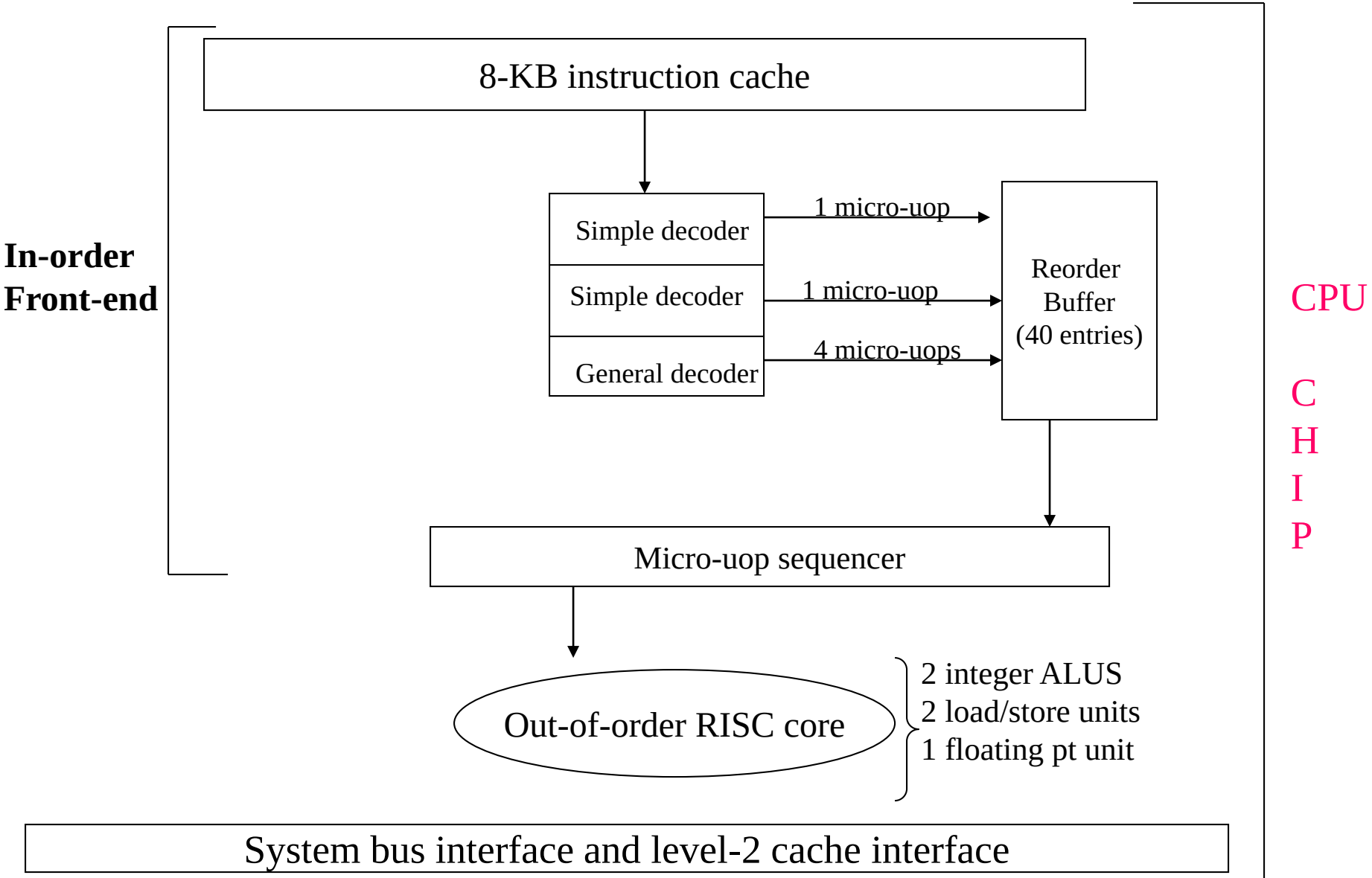
Instructions for execution are retrieved from only one Queue, say A. Now, when a branch instruction is predicted as taken, then the current instruction queue is frozen and instructions are fetched from the branch target address and are placed in Queue B.

If the prediction is correct, Queue B now becomes operational; else instructions are taken from Queue A.

BTB has 256 66-bit entries – 32 bits branch instr addr, 32 bits branch dest.addr and 2 bits history)

Intel Pentium directly executes x86 CISC instructions, but internally the chip implements a *de-coupled CISC/RISC architecture* as shown in the figure (see the next slide)

At the front-end, three x86 instructions can be *decoded in parallel* by an *in-order translation engine*. These decoders translate the x86 instructions into 5 RISC-like micro-operations, denoted as *µops*. Each of the two simple decoders produces one *µop*, while a general decoder converts a complex instruction into 1 to 4 *µops*.



At the back-end, a superscalar execution engine is capable of executing *five µops out-of-order* on five execution units in the RISC core. The five execution units are:
2 integer ALUS, 2 load/store units and 1 floating point unit. These *µops* are first passed to a 40-entry reorder buffer shown, where they wait for the required operands become available.

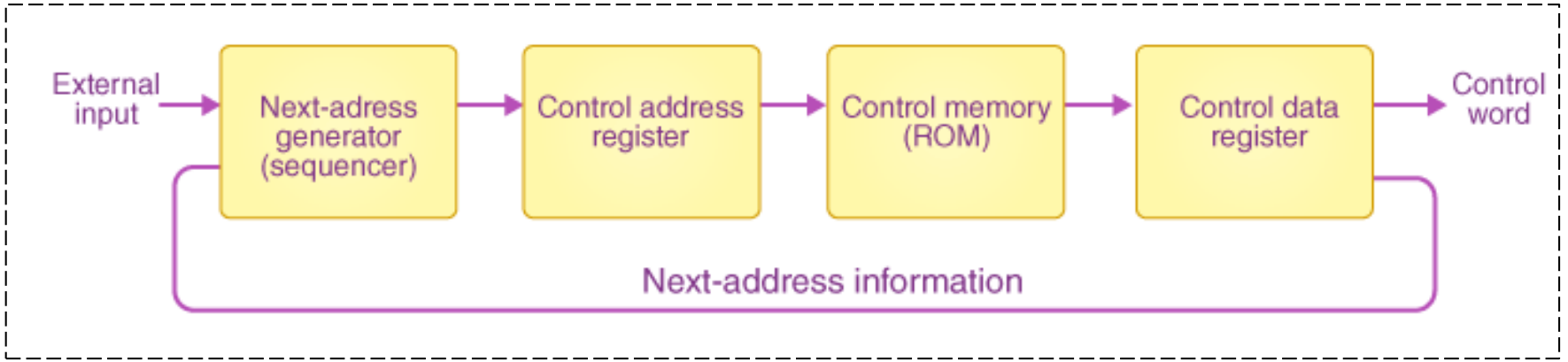
From this reorder buffer, the *µops* are issued to a 20-entry reservation station (RS) [*entry point of the RISC core - not shown in the figure*], which queues them until the needed execution unit is free.

This design allows *µops* to execute *out-of-order*, making it easier to keep parallel execution resources busy. At the same time, the fixed-length *µops* are easier to handle in speculative, out-of-order core than complex, variable-length x86 instructions.

Memory hierarchy: A special design is carried out by Intel for this processor. A level-2 cache chip that is mounted in the same package with the CPU chip. Direct connections exists between the CPU and the cache chips. The level-2 cache delivers 64 bits per cycle, even with a 200-MHz clock. The on-chip caches are totalled 16KB, split into 8KB for instr and 8 KB for data

CISC CPU - Hardwired Control - Micro-Programmed Control Mechanism

Micro-Program Control Unit (MPCU)



Software approach is used to implement a MPCU;

Program made up of micro-instructions is used to carry out a series of micro-operations

MPCU is also called as a *Micro-program sequencer*

Working Sequence:

Micro-instruction address is specified in the control memory address register;

All the control (signals) information in the form of words is saved in the control memory (ROM)

Micro-instruction received from memory is stored in the control register;

A control word in the micro-instruction specifies one or multiple micro-operations for a data processor.

The next address is calculated in the circuit of the next address generator and then transferred to the control address register for reading the next micro-instruction when the micro-operations are being executed.

Because it determines the sequence of addresses received from control memory, the next address generator is also known as a microprogram sequencer.