

Chapter 3 Processor and Memory Technology

Contents of this chapter

- Design space of modern day computers
- Instruction pipelines
- Various instruction set architectures
- CISC scalar processors
- RISC scalar processors

- Superscalar processors & Pipelining in superscalar processors
- VLIW Architecture
- Multi-threaded Processors
- Message Passing and Shared Memory programming examples
- Memory hierarchy technology - Important properties
 - Inclusion, coherence, and locality
- Memory capacity planning - the design of memory hierarchy
- Combining all that we have learnt so far - A **GPU** Architecture & Programming!

Reference : Kai Hwang's book

Pages: 157-165; case study-pages 167-169;
169-170; case study-pages 174-176; 177-184;
188-196;

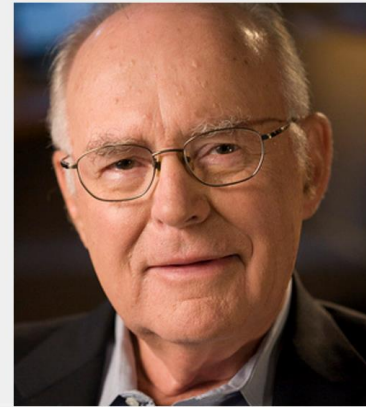
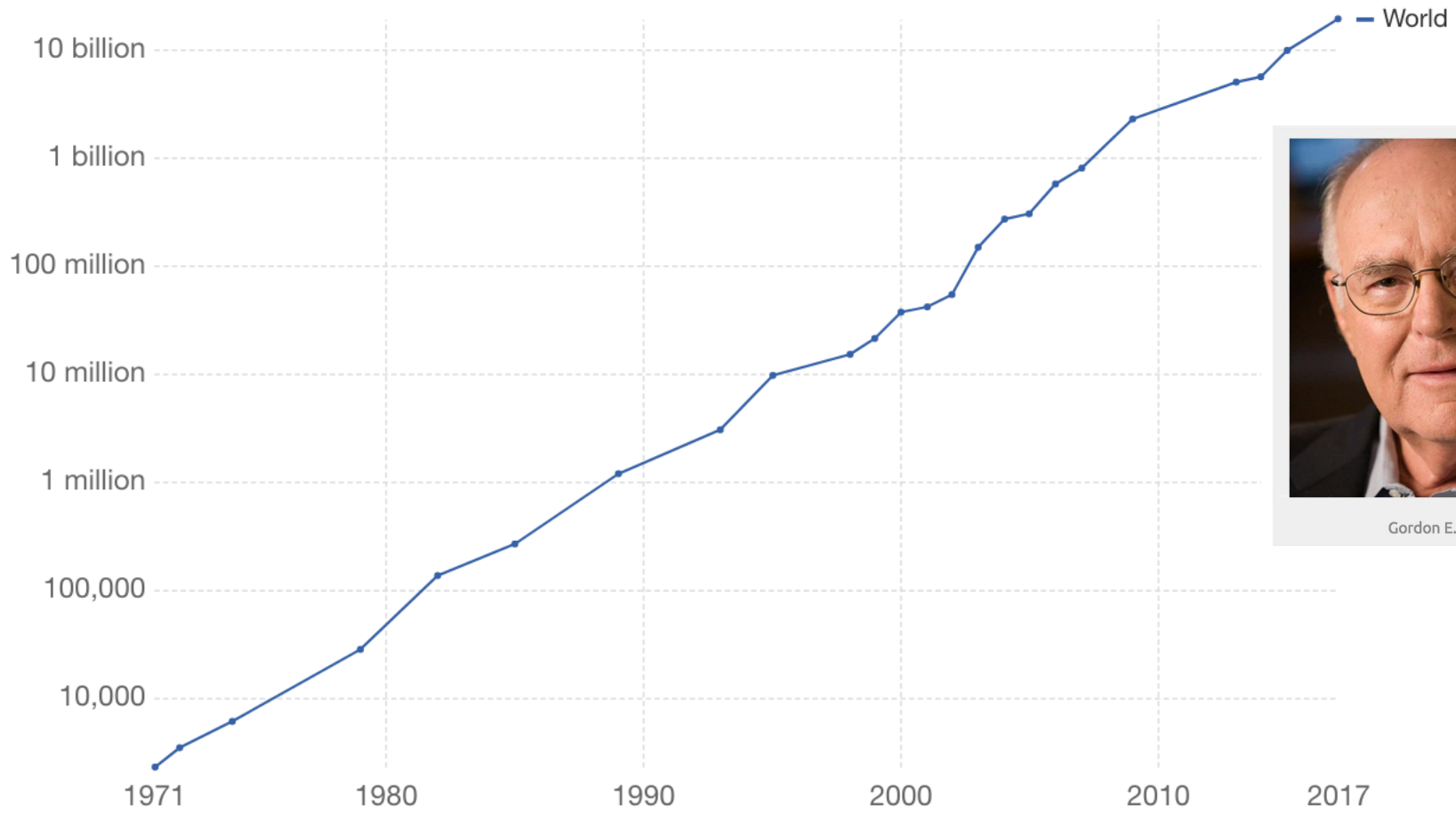
Case study on intel Pentium Pro Processor - Use the slides
of this presentation + Intel 80X86 family + [IA-64](#) (brief)

- *I and D cache allocation - pdf notes available on web*
- *RoR model*

Some performance indicators...

Moore's Law: Transistors per microprocessor

Number of transistors which fit into a microprocessor. This relationship was famously related to Moore's Law, which was the observation that the number of transistors in a dense integrated circuit doubles approximately every two years.



Gordon E. Moore

Source: Karl Rupp. 40 Years of Microprocessor Trend Data.

OurWorldInData.org • CC BY-SA

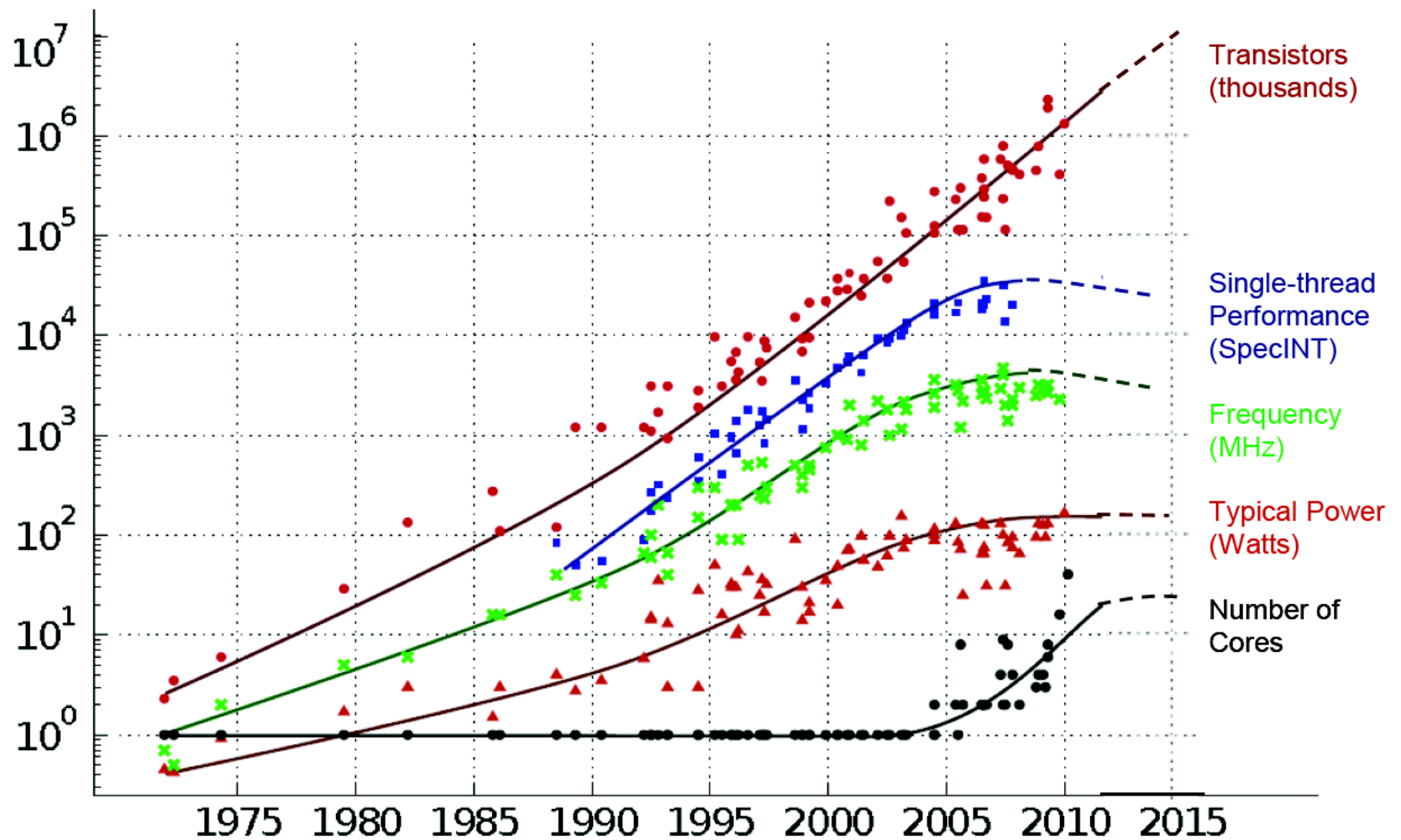
Moore's Law is a computing term which originated around 1970; the simplified version of *this law states that processor speeds, or overall processing power for computers will double every two years.*

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 transistors than to speed.

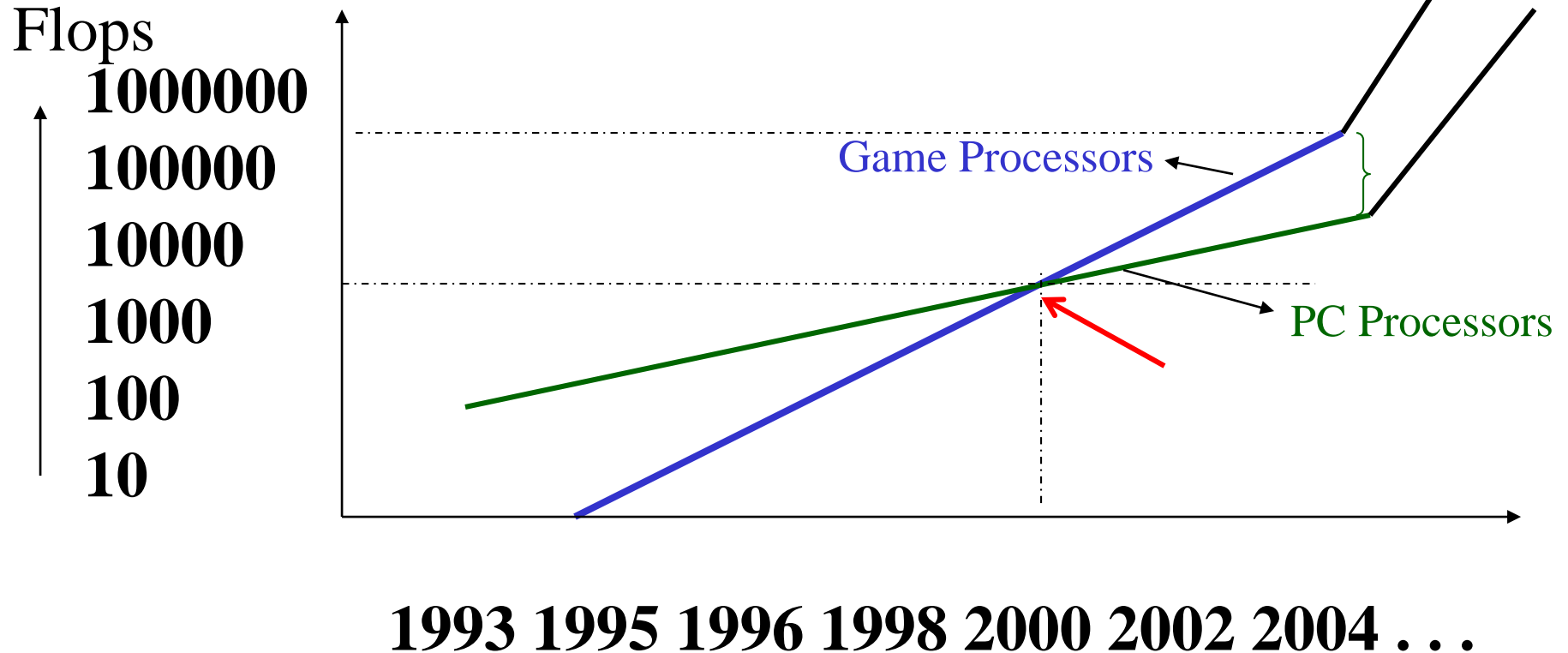
35 YEARS OF MICROPROCESSOR TREND DATA



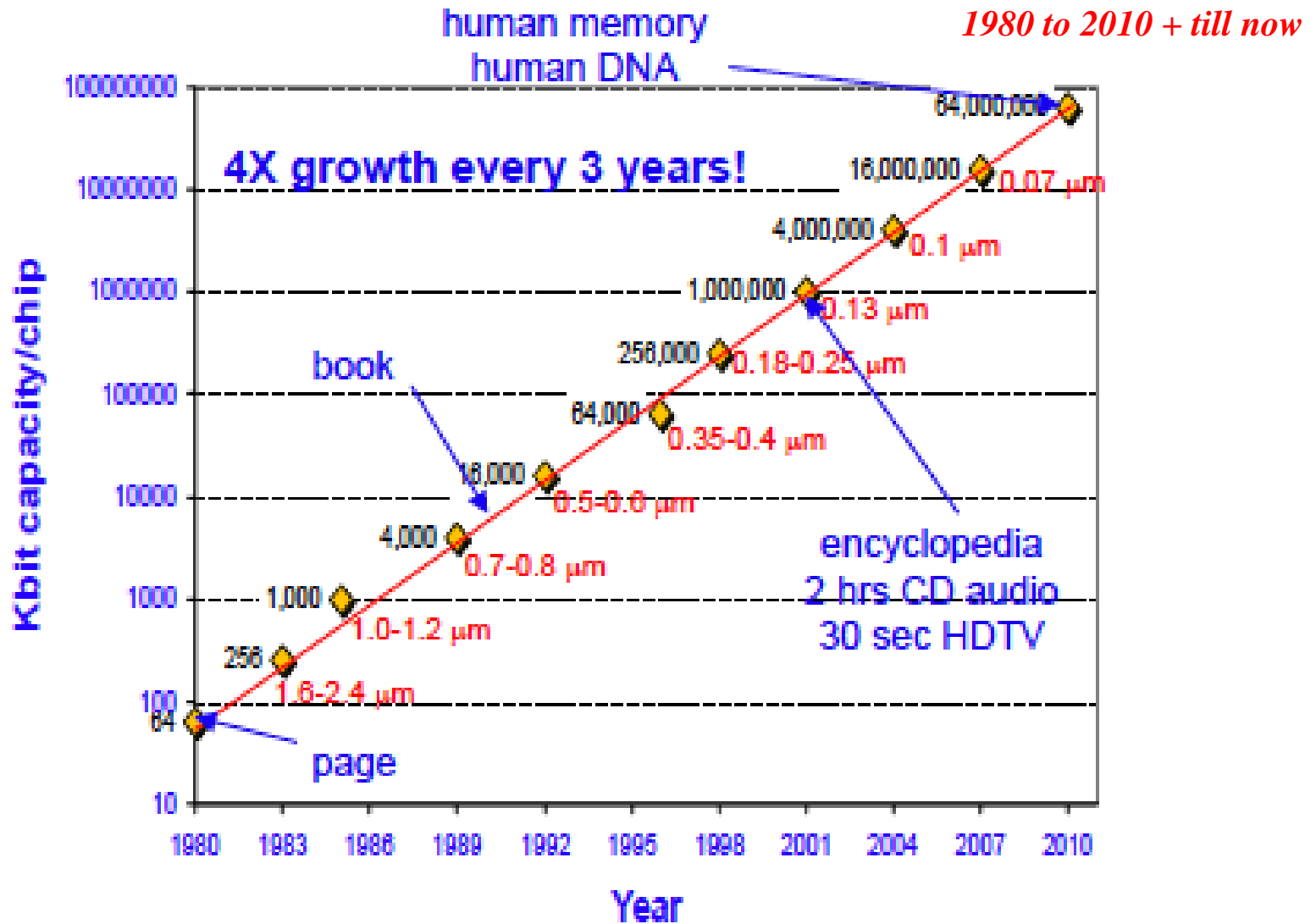
Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Game CPUs vs Conventional CPUs Performance over Time

(Game processors take the lead on media performance)



Similar Story for Storage Trends (DRAM Capacity)...



Some useful remarks

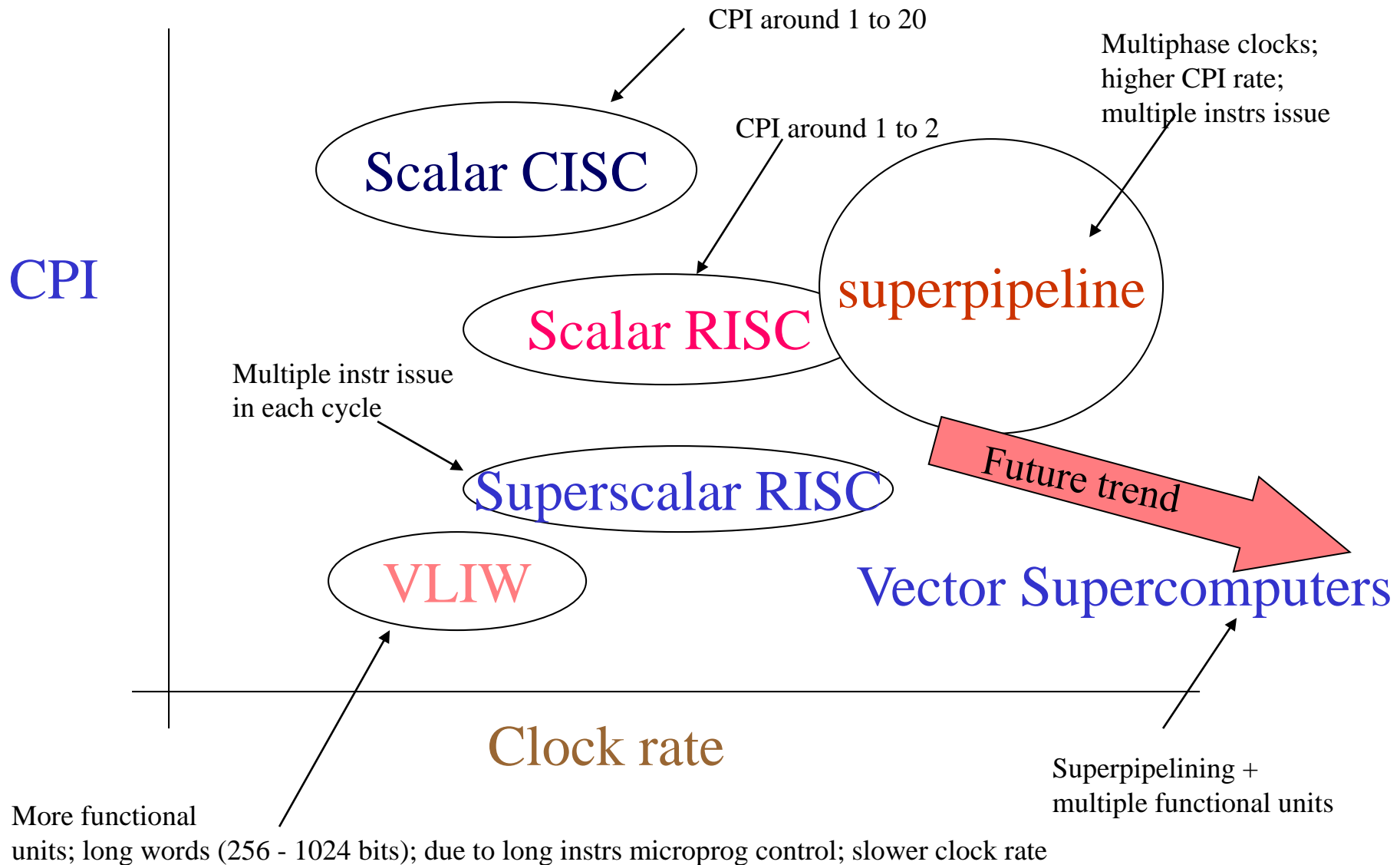
- Power grows with number of transistors and clock frequency
- Power grows with voltage; $P = CV^2f$
- Going from 12V to 1.1V reduced power consumption by 120x in 20 yrs
- Voltage projected to go down to 0.7V in 2018, so only another 2.5x
- Power per chip peaking in designs:
 - - Itanium II was 130W, 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/8600, 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 - Intel i860, SPARC, MIPS R3000, 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.

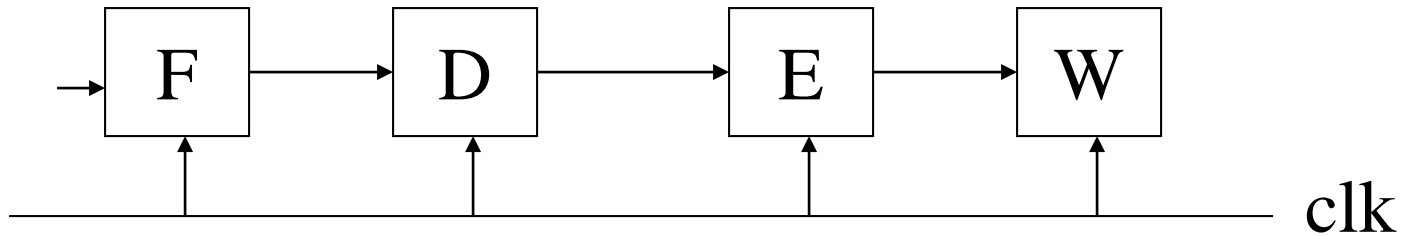
Instruction pipelines

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

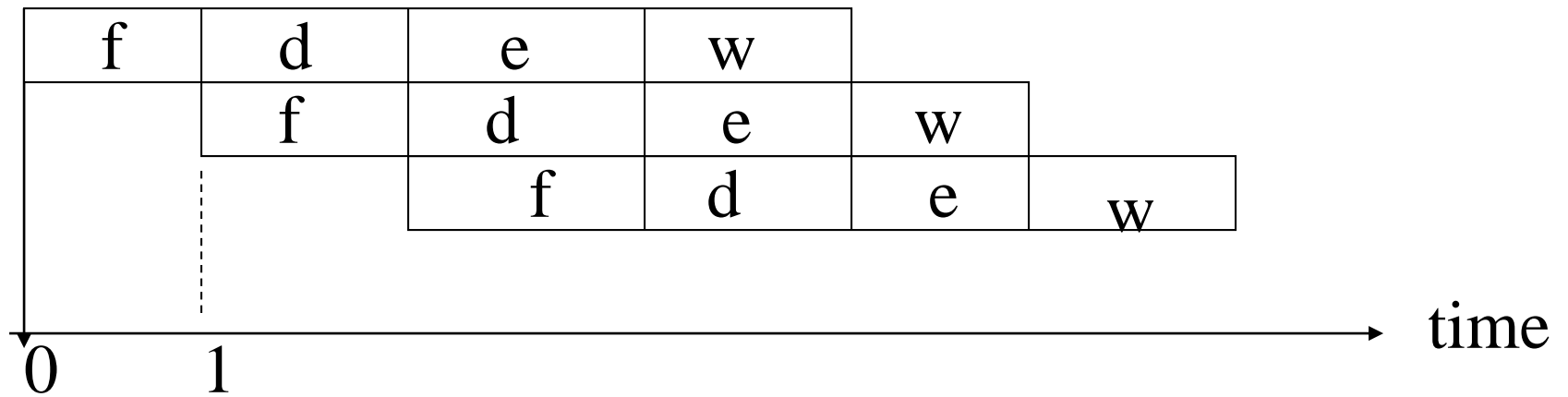
5. Resource conflicts : Demand for the same functional unit at the same time.

We will see all the design issues in the pipeline chapter

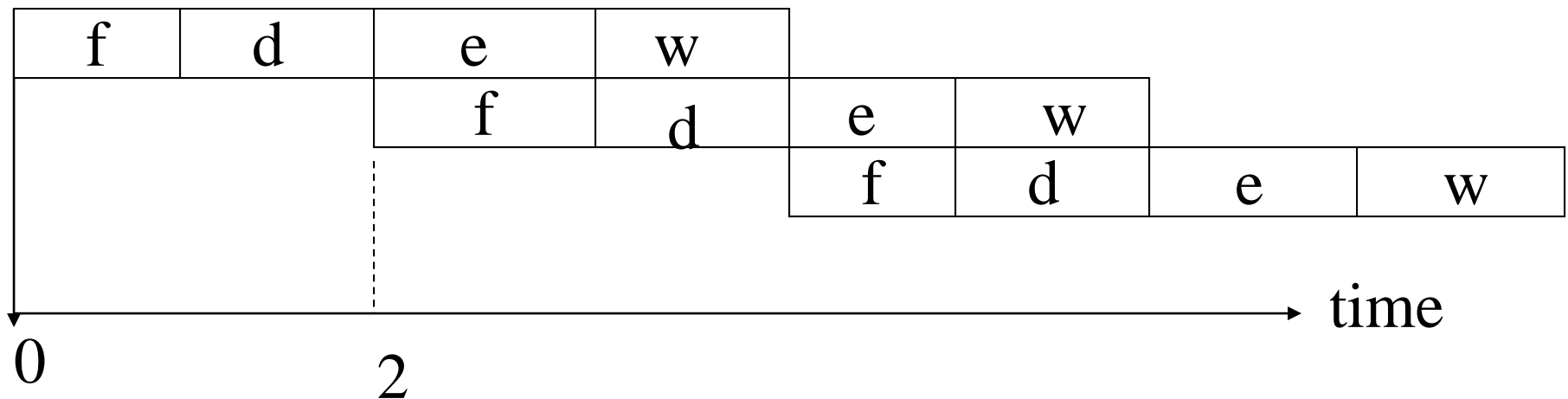
A base scalar processor is defined as a machine with

- 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 $0 + R2$

I2: ADD R1, R1, #1 ; $R1 = R1 + 1$

I3: ST 0(R2), R1 ; Store R1 at address $0 + R2$

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.

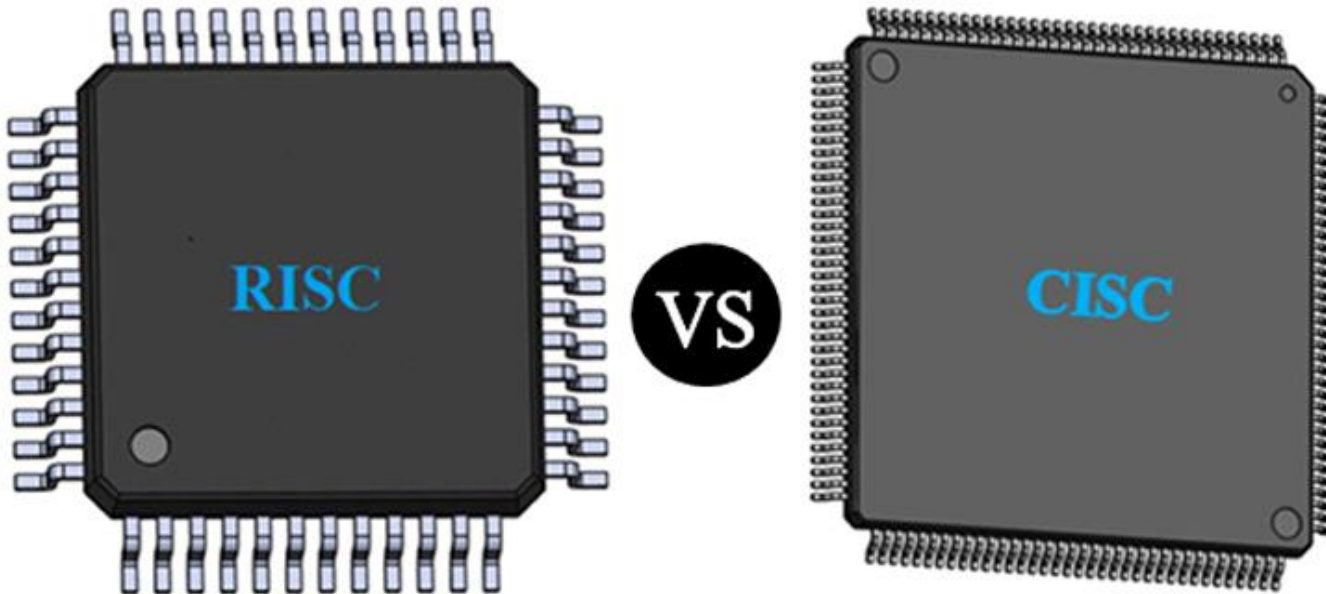
The complexity of an IS is attributed to the following:

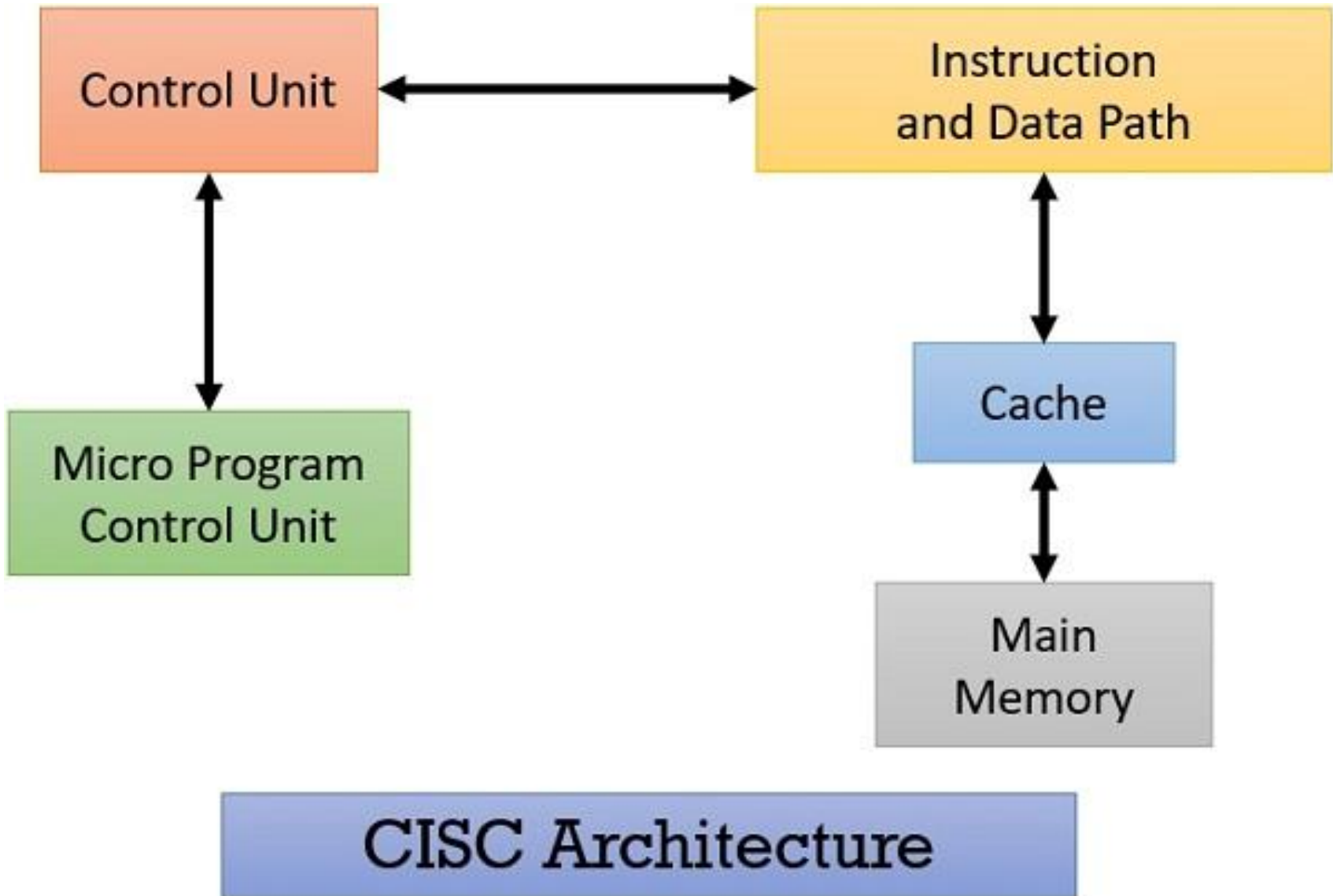
Instruction formats, data formats, addressing modes, general-purpose registers, opcode specs, and flow control mechanisms.

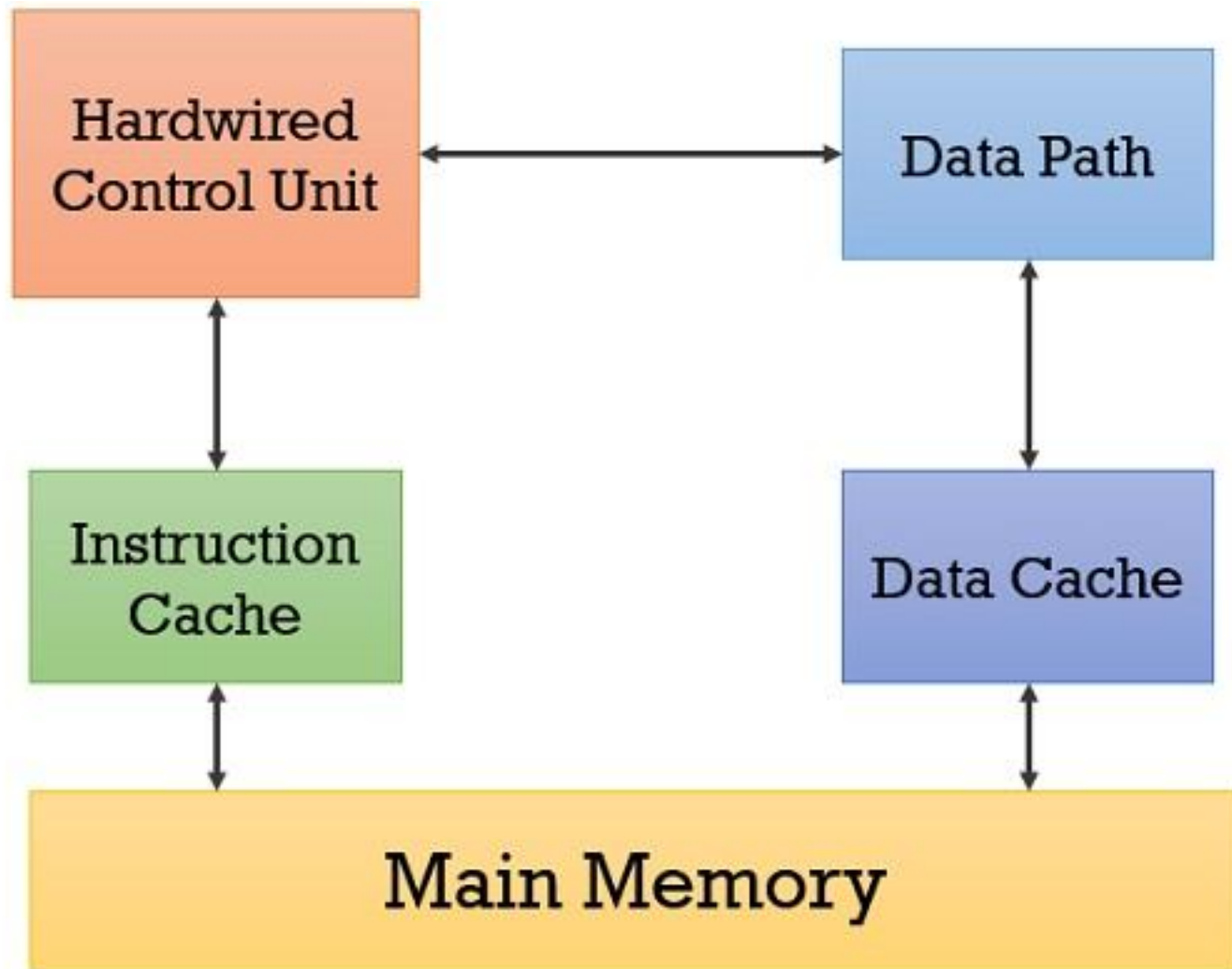
Based on the experience in the design of a processor, two different architectures have been proposed:

CISC (Complex instruction Set computing)

RISC (Reduced Instruction Set computing)







RISC Architecture

Architectural Distinctions

Characteristics

- IS/formats
- Addressing modes
- GPRs

CISC

Large sets with variable formats; 16-64bits/Inst

120-350 instructions typically

12-24

8-24; data+inst
in a single cache

RISC

Small set
with fixed
formats(32-bits)

Less than 100 instructions

3-5

32-192
split type cache

Characteristics

CISC

RISC

• Clock rate/CPI	$x / 2$ to 15	$> k.x, \quad k > 1 /$ $CPI < 1.5$
• 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,

$$\text{Register file size} = (L+C)W + G$$

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, Superpipeline processors, & Superscalar Superpipeline 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.

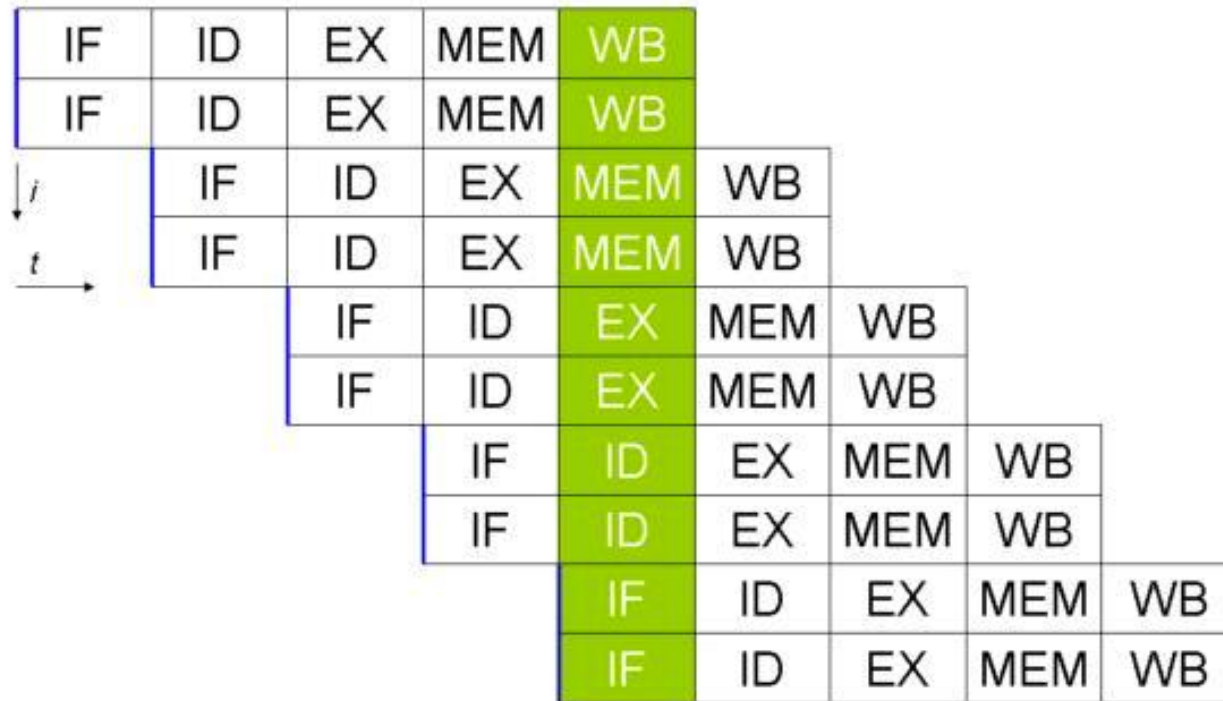
Superscalar processors belong to the class of processors that are designed exclusively to exploit more *instruction-level parallelism* in the user programs.

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

Pipelining in superscalar processors: (*refer to fig next slide*)

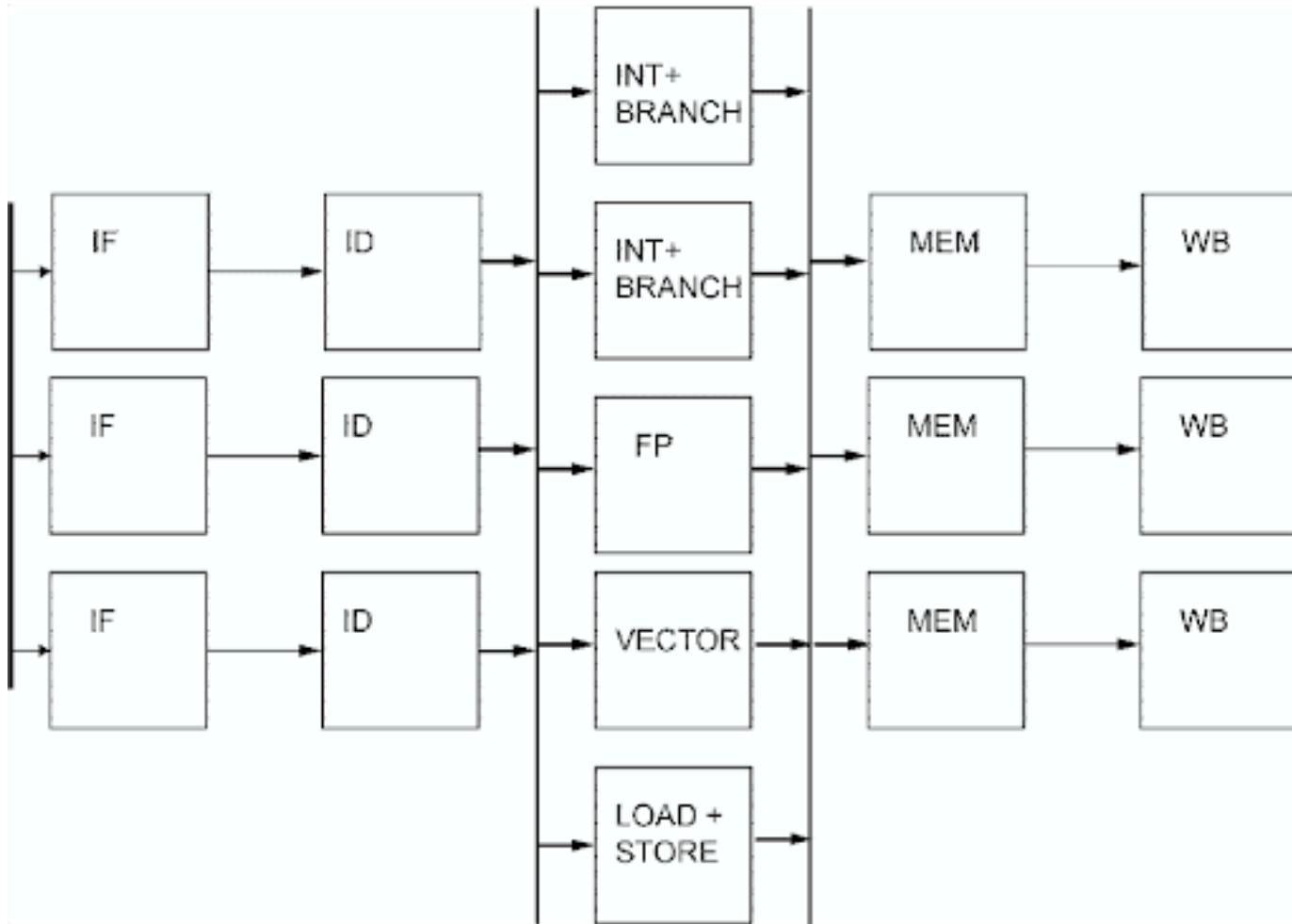
In general, a superscalar processor of degree m can issue m instructions per clock cycle. In order to fully utilize its power, m instructions must be executed in parallel.

Two-issue Super Scalar CPU



Degree of super scaling $m = 2$

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



Superpipeline Processors

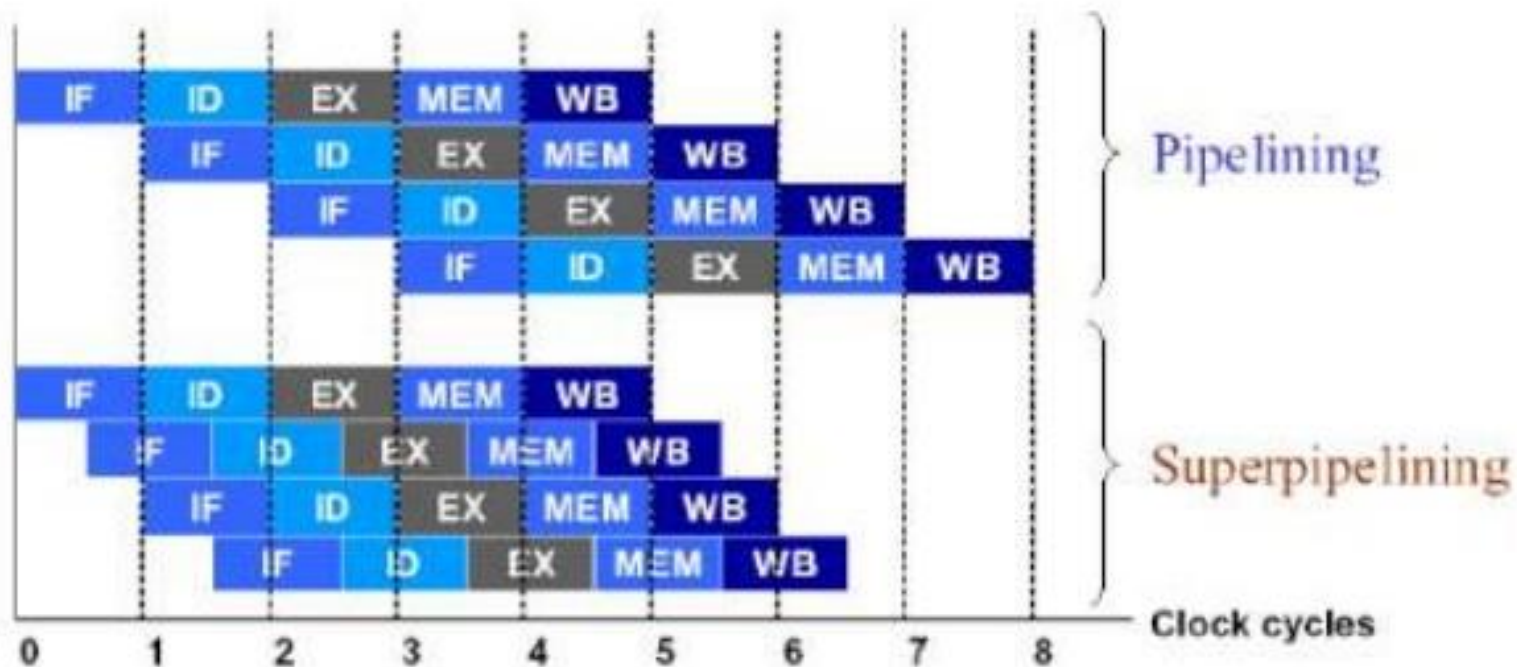
Extension of pipelining idea here! In normal pipelining, each of the stages takes the same time as the external machine clock. But, not all pipeline stages need the same amount of time. For example, *instruction decoding (especially in a RISC machine) is faster than the other stages such as fetches and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

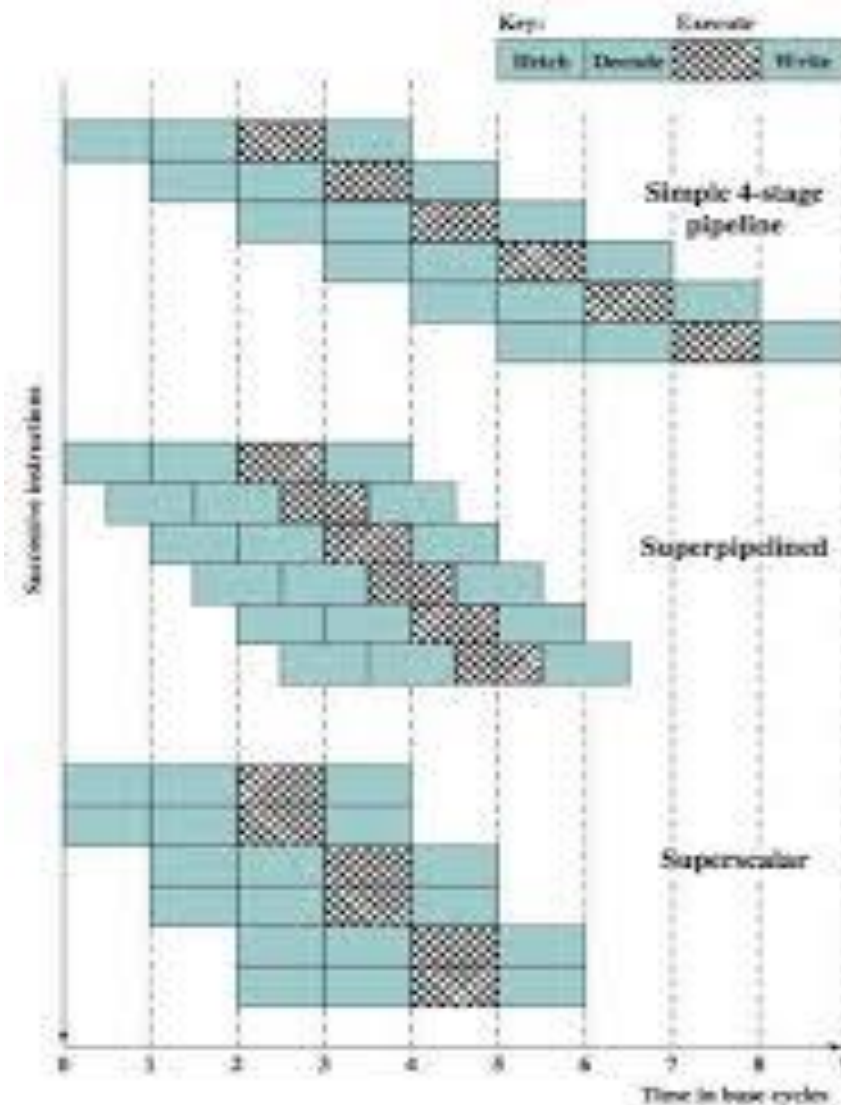
Super pipeline Performance



- The performance is shown below in the figure:



Superscalar versus Superpipelining



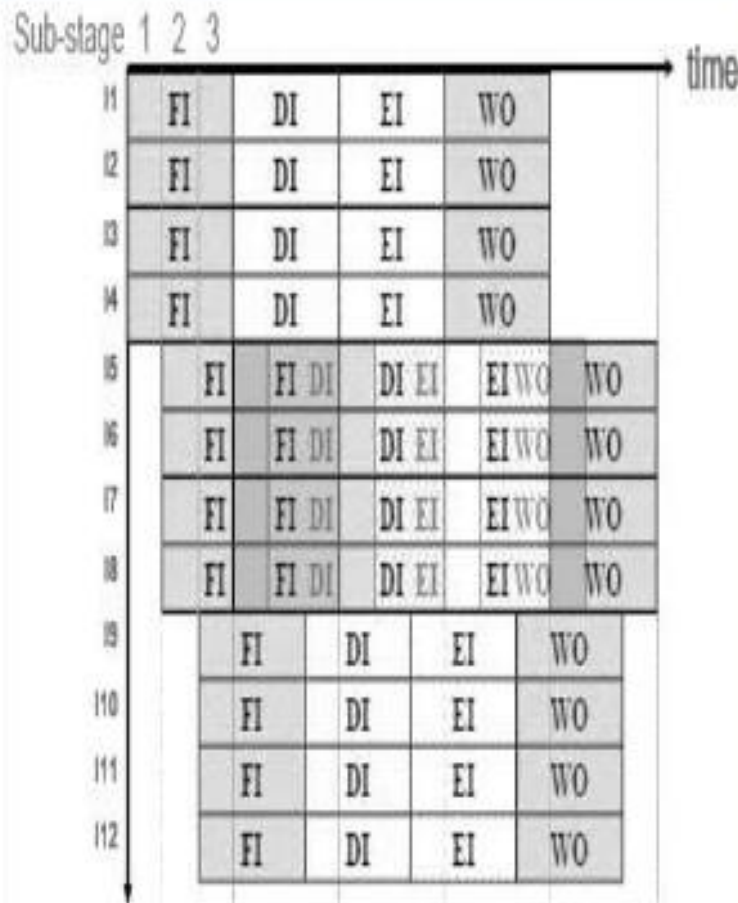
Degree=2

Degree=2

The timing diagram shown above is for an ideal situation and this may not be always true in all clock cycles, as some instructions may cause pipeline stalling.

Superscalar processors largely depend on exploiting parallelism through compilers. (See Example 4.5, page 181-182 - *Reading assignment*)

Superpipelined Superscalar



*Superpipeline
degree: 3
Superscalar
degree: 4*

*Compared to a base
machine, this has
12 times Speed-up.*

VLIW Architecture

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

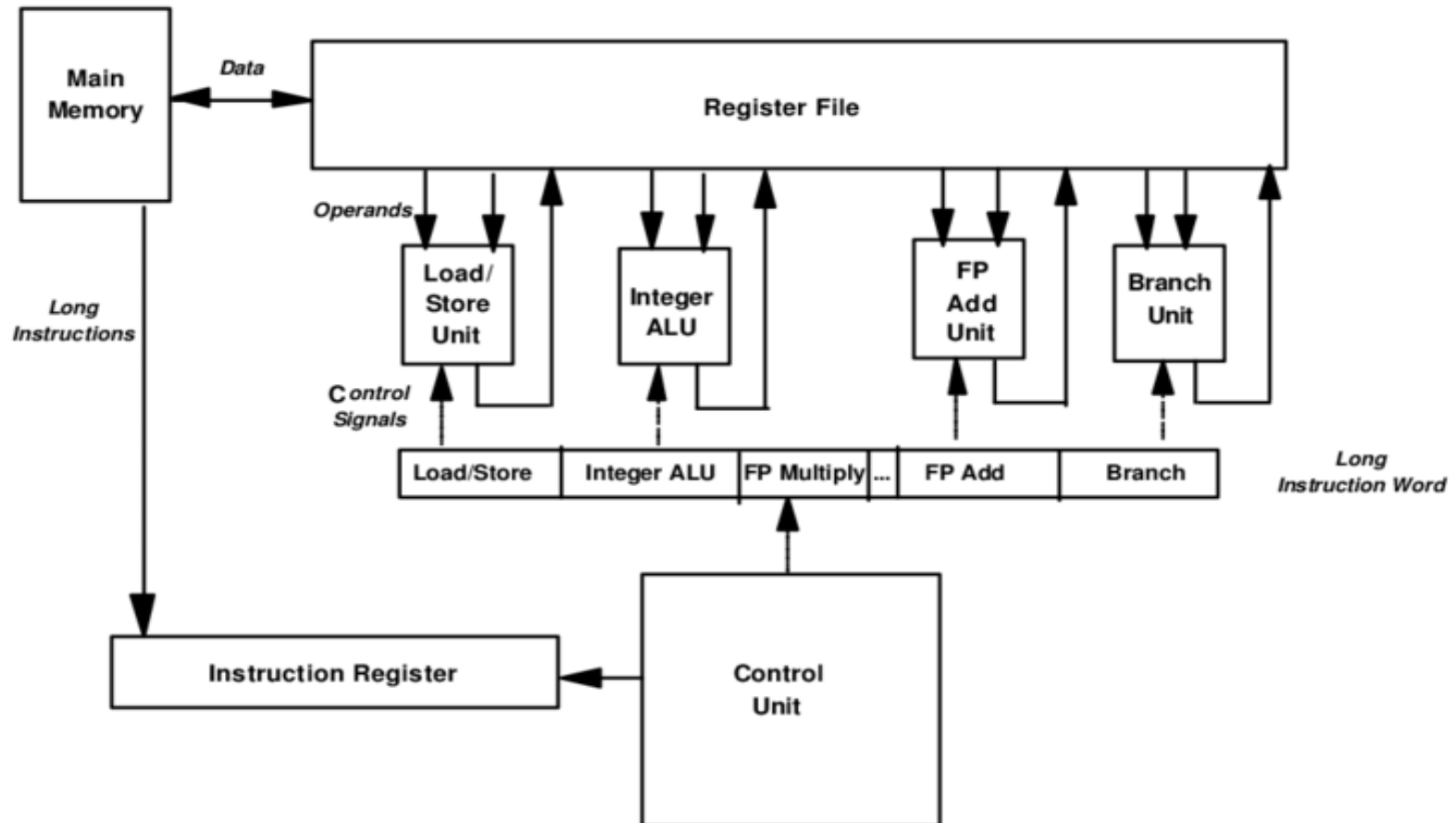
A VLIW machine has instruction words hundreds of bits in length.

(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)

$$\text{Computation: } y = a_1 * x_1 + a_2 * x_2 + a_3 * x_3$$

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 a2	Load a3
Load x1	Load x2	Load x3
	MUL z1,a1,x1	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 nonparallel 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)

- Multi-threaded Processors
- *Shared Memory Application Example on Multi-threaded processors* (Refer to a separate doc provided)

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

A, I1	FI	DE	EX	MEM	SR				
A, I2		FI	DE	EX		MEM	SR		
A, I3			FI	DE		EX		MEM	SR
.....									
A, I8	(encounters a long latency operation during execution)								

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?

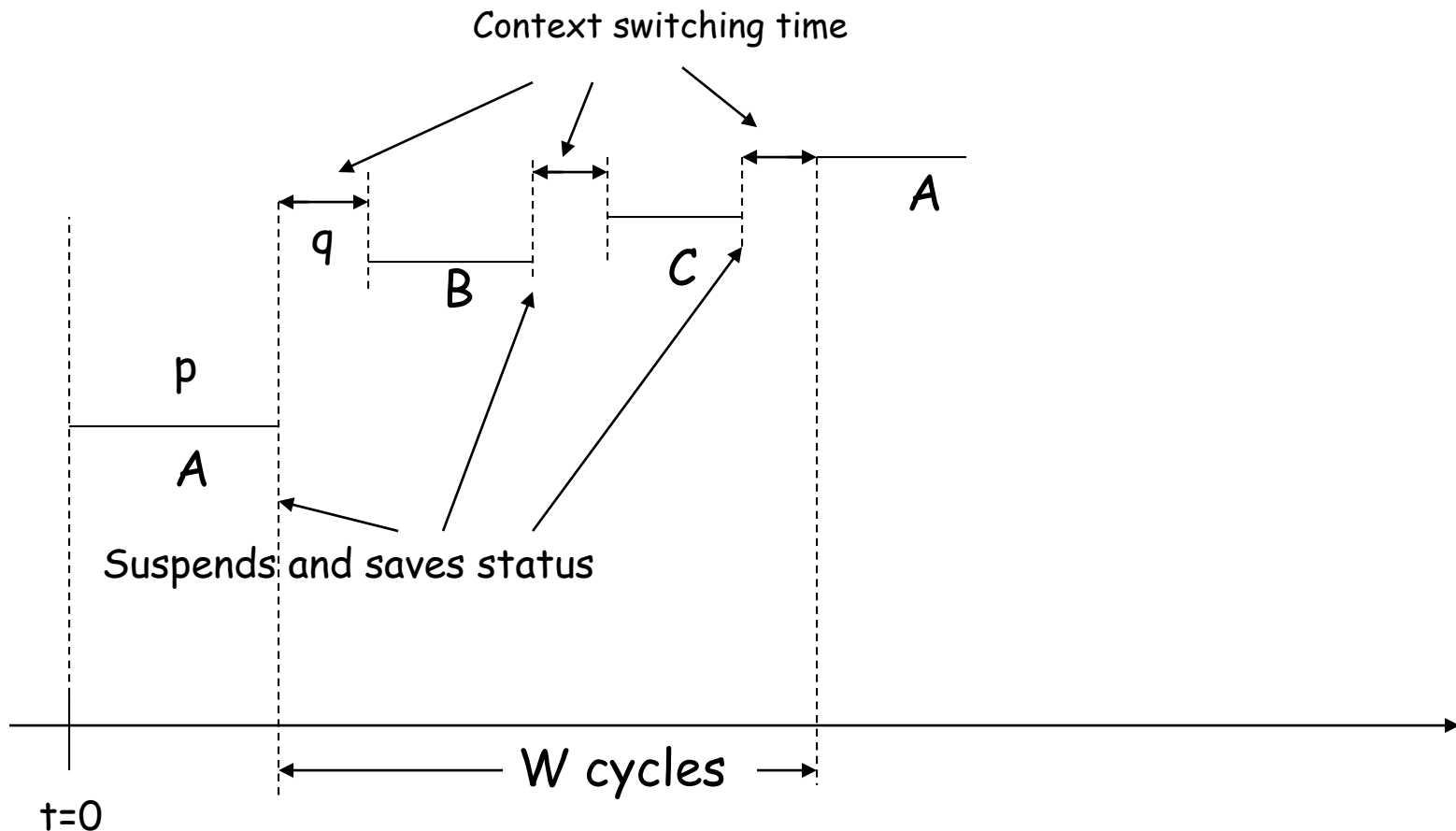
In order to answer these questions, we must identify the parameters which affect the performance of the processor. These are as follows.

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.

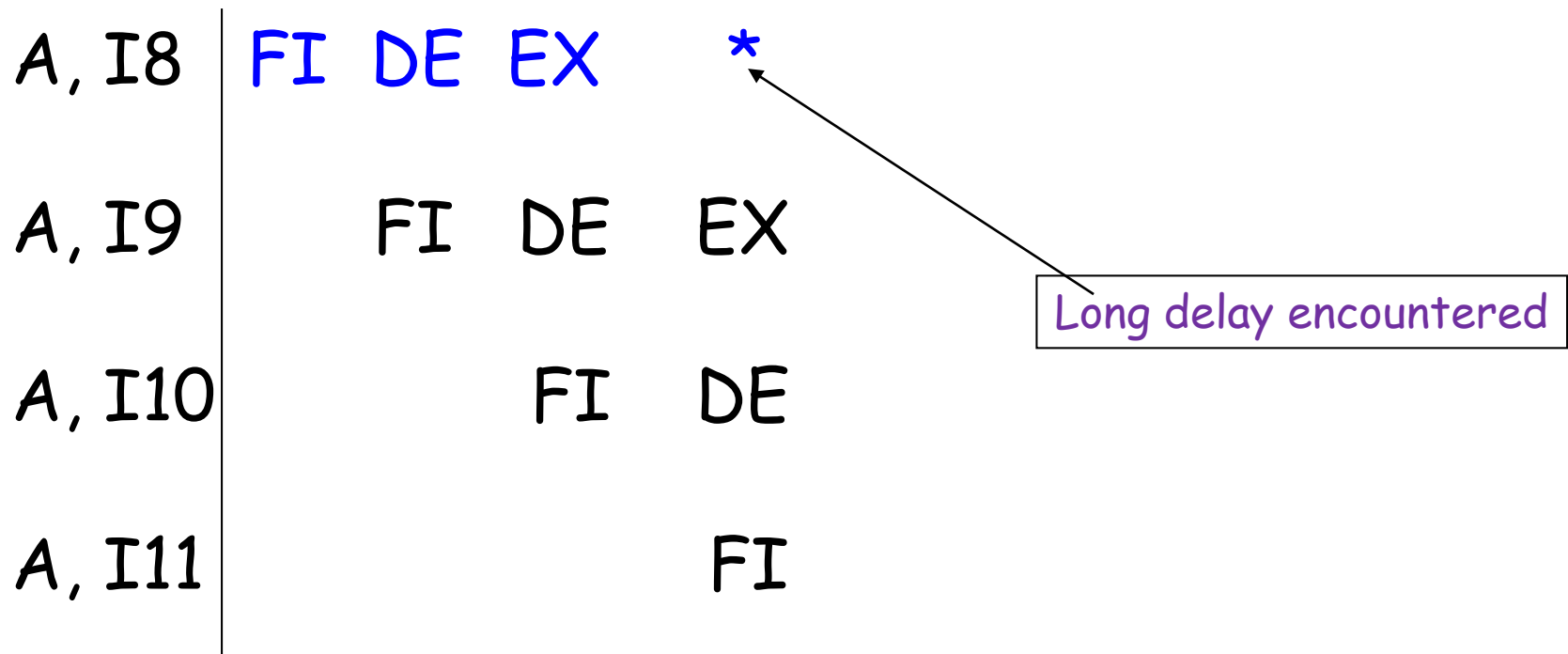


$nq + (n-1)p \leq w$ This implies that the number of threads that can be supported is

$$n \leq (p+w)/(p+q)$$

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 $\sim 88\%$ 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 $(d + w \cdot q)$, 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.

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

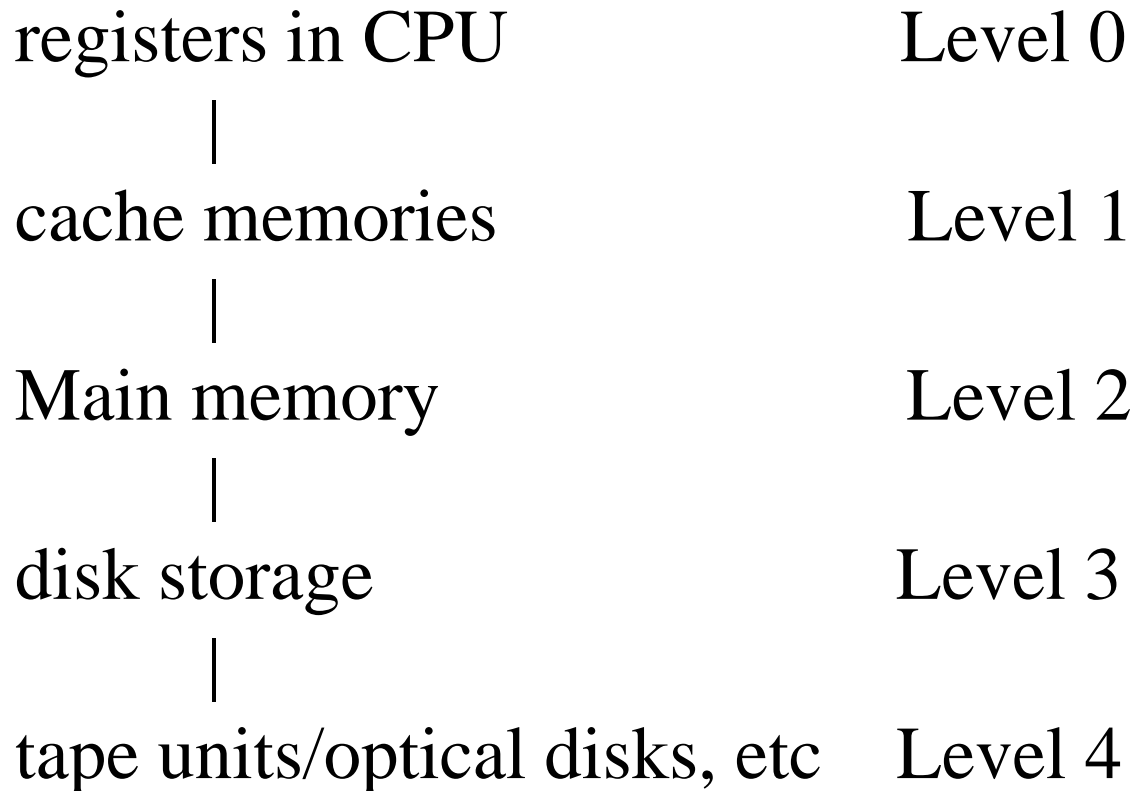
A, I1	FI	DE	MEM	EX	SR
B, I1	FI	DE	MEM	EX	SR
C, I1	FI	DE	MEM	EX	SR
D, I1	FI	DE	MEM	EX	SR
E, I1		FI	DE	MEM	EX SR
F, I1		FI	DE	MEM	EX SR

Motivation is to combine multithreading and superscalar features - Happening in GPU architectures now!

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

Memory Hierarchy Technology

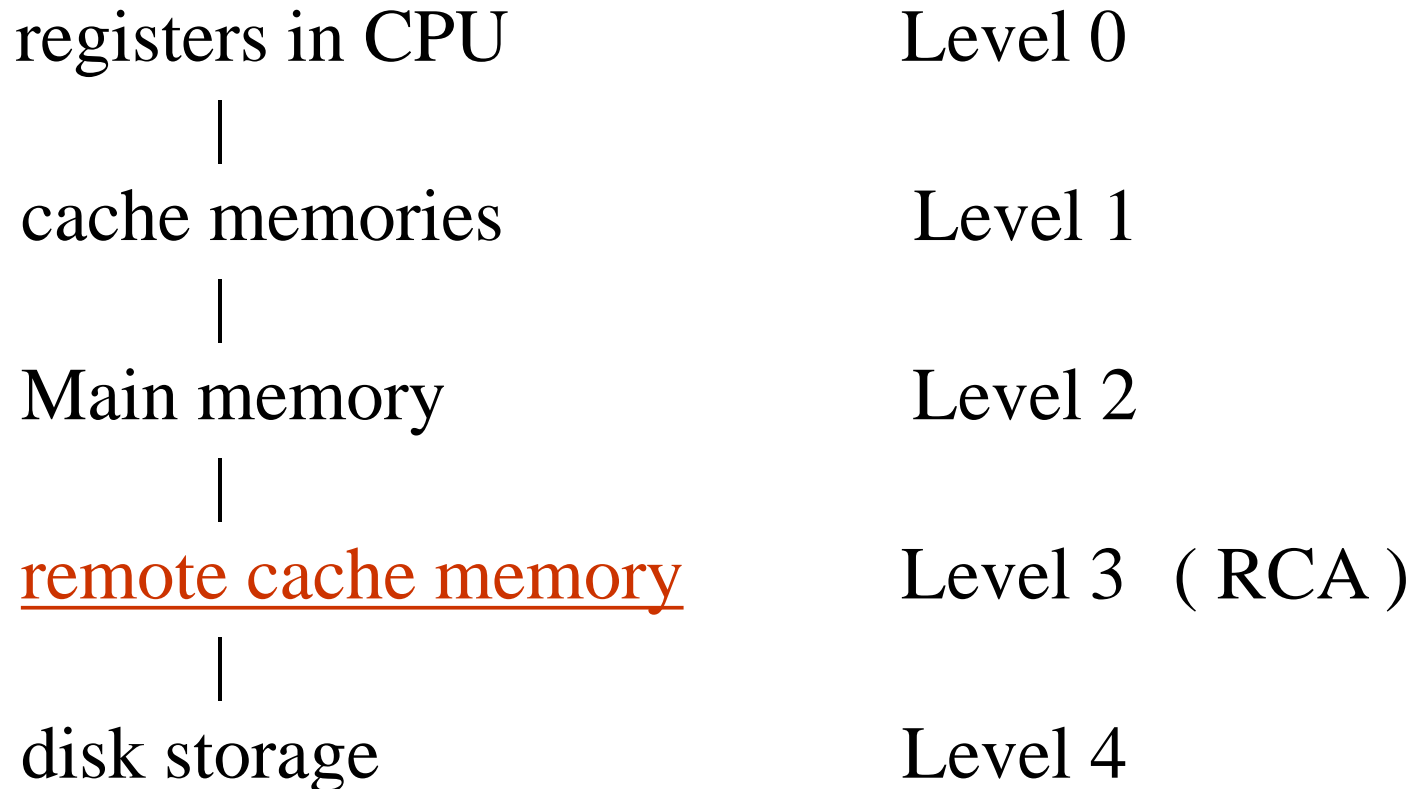
Storage devices hierarchy can be classified as follows.



The memory technology at each level is characterized by the following five parameters.

1. Access time (t_i)
2. Memory size (s_i)
3. Cost per byte (c_i)
4. Transfer bandwidth (b_i)
5. Unit of transfer/granularity (x_i)

A small digression here! (*This slide not in syllabus!!*)



In general,

$t_{i-1} < t_i$, $s_{i-1} < s_i$, $c_{i-1} > c_i$, $b_{i-1} > b_i$, $x_{i-1} < x_i$, for all $i=1,2,3,4$.

Details of a typical mainframe (1993) can be found in Table 4.7, page 190

Some useful properties in the design of memory technology

1. Inclusion
2. Coherence
3. Locality of References

Let the memory hierarchy be denoted as M_1, M_2, \dots, M_n and consider the cache memory M_1 , which directly communicates to CPU registers. The outermost level M_n contains all the information, and this level forms the virtual address space of memory hierarchy effectively.

Inclusion Property: This property is stated as

M_i is a proper subset of M_{i+1} , for all $i=1, 2, \dots, n-1$.

During processing subsets of M_i are copied to M_{i-1} and so on. If an information is found in level i , then the information can also be found in all upper levels $i+1$ to n .

Information transfer between CPU and cache is in terms of words (typically 4 or 8 bytes depending on the word length of the machine). **Refer to Fig. 4.18 - page 191**

Pages are the fundamental units that are transferred between MM and disks. Data transfer between disk and the tape units are in terms of file segments.

Coherence Property: This property emphasizes the need for the copies of same data to have same information at the levels wherever the data is currently residing. This is also referred to as *consistency* property. If a word is modified in the cache, then it must be updated at all levels.

Two strategies are followed to update, in general.

1. Write through: This demands immediate updates in a higher level memory, if a word is modified in the immediate lower level memory.
2. Write back: This method delays the update in a higher level memory until the word being modified is replaced or removed from the immediate lower level memory.

LRU replacement policy will be studied in detail later in next chapter.

Locality of references : The memory hierarchy basically originated by studying the program behaviour extensively. The behaviour refers to the “pattern” of access and the frequency of use of a page/word in the memory.

The access pattern tends to be clustered in certain regions in *time, space, and ordering*.

90-10 rule by Hennessy and Patterson (1990) states that a typical program may spend 90% of its execution time on only 10 % of the code such as the innermost loop of a nested looping operations.

Refer to Fig. 4.19 on page 192 for one such pattern

Memory reference patterns are caused by the following locality properties:

1. Temporal: Recently referenced items are likely to be referenced in the near future
2. Spatial: Refers to the tendency for a process to access the items whose addresses are near to one another.
3. Sequential: In typical programs, execution of instructions follow an order unless any branch instructions are met.

The sequentiality also contributes to spatial locality as sequentially coded instructions and array elements are often stored in adjacent locations. Each type of locality affects the design.

The temporal locality leads to LRU algorithm; spatial locality helps in determining the size of the data units to be transferred between memory units; and the sequential locality helps to determine the grain size for optimal performance.

Working sets : the subset of pages or addresses referenced within a given window of time is referred to as WSs.

Note that WSs are function of time, however, maintains some sort of continuity. The time window size is often crucial in the design of the size of the cache.

Memory Capacity Planning

The performance is determined by *effective access time* T_{eff} to any level in the hierarchy.

Hit ratios: When a memory M_i is accessed and if the desired word is found, it is referred to as a *hit*, otherwise it is considered as a *miss*.

The hit ratio (h_i) is the probability that a word/information will be found when accessed in M_i . It is a function of the characteristics of the adjacent levels. Obviously, the miss ratio is $1-h_i$.

The hit ratios at successive levels are a function of memory capacities, management policies, and program behaviour. In the analysis, successive hit ratios are *independent* random variables assuming values between 0 and 1.

Further, we assume that $h_0=0$ and $h_n=1$. This means that the CPU always access M_1 first and access to the outermost level is always a hit.

Access frequency at a level i is defined as

$$f_i = (1-h_1)(1-h_2)\dots(1-h_{i-1}) h_i$$

This is nothing but the probability of accessing the level M_i amidst $(i-1)$ misses at the lower levels and hit at i . Note that $f_1 + f_2 + \dots + f_n = 1$ and $f_1 = h_1$.

Due to the *locality property*, the access frequencies decrease rapidly from the lower levels, i.e., access freq at level i is greater than $i+1$. This means that the inner levels are accessed more often than the outer levels.

Effective Access Time is defined as

$$T_{\text{eff}} = h_1 t_1 + (1-h_1)h_2 t_2 + (1-h_1)(1-h_2)h_3 t_3 + \dots + (1-h_1)(1-h_2)\dots(1-h_{n-1})h_n t_n$$

Hierarchy Optimization The total cost of a memory hierarchy is estimated as follows:

$$C_{\text{total}} = c_1 s_1 + c_2 s_2 + \dots + c_n s_n$$

Since $c_i > c_{i+1}$, we have to choose $s_i < s_{i+1}$. *The optimal design should result in a T_{eff} close to t_1 of M_1 and a total cost close to c_n of M_n*

In practice, it is difficult to achieve this objective. This problem is formulated as a linear programming problem. That is,

$$\min T_{\text{eff}}, \text{ subject to: } s_i > 0; t_i > 0, i=1, \dots, n \text{ and} \\ C_{\text{total}} < C_0$$

Refer to an example (next slide)

Note: You will study about the techniques used in managing the memory capacity (in MM and cache) in Chapter 4, in detail.

Memory Hierarchy (Cont'd)...

Example: Consider a 3 level memory hierarchy for an electronic device that has limited computing power CPU with the following specifications for memory characteristics.

Memory Level	Access time	Capacity	Cost/byte
Cache	$t_1 = 25\text{nsecs}$	$s_1 = 512\text{Kbytes}$	$c_1 = \$1.25$
Main memory	$t_2 = ?$	$s_2 = 32\text{Mbytes}$	$c_2 = \$0.2$
Disk	$t_3 = 4 \text{ msecs}$	$s_3 = ?$	$c_3 = \$0.0002$

The design goal is to achieve an effective memory access time $t = 10.04$ micro-secs with a cache hit ratio of $h_1 = 0.98$ and a hit ratio of $h_2 = 0.9$ in the main memory. Also, the total budget given is upper bounded by \$15,000.00.

Memory Hierarchy Example (Cont'd)...

Solution:

The memory hierarchy cost can be computed as:

$$C = c_1 s_1 + c_2 s_2 + c_3 s_3 \leq 15000$$

The maximum capacity of the disk is thus obtained as $s_3 = 39.8$ GBytes without exceeding the budget.

Next, we want to choose a RAM memory after computing the access time. Using the expression for effective access time, we obtain,

$$t = h_1 t_1 + (1-h_1)h_2 t_2 + (1-h_1)(1-h_2)h_3 t_3 \leq 10.04$$

Substituting all the known parameters, we obtain $t_2 = 903$ nsecs

Memory Hierarchy Example (Cont'd)...

Some points to think!

*Q1: Suppose one wants to double the main memory capacity at the expense of reducing the disk capacity under the same constraints. This change **WILL** / **will not** affect the cache hit ratio.*

Q2: In the above Q1, it may increase the hit ratio in the main memory. (True/False)?

Q3: In the above Q1, the effective access time will be enhanced. (True/False)?

Combining the concepts of SM / Multi-threading...

GPU architecture

- CPU + GPU combo! (Why?)

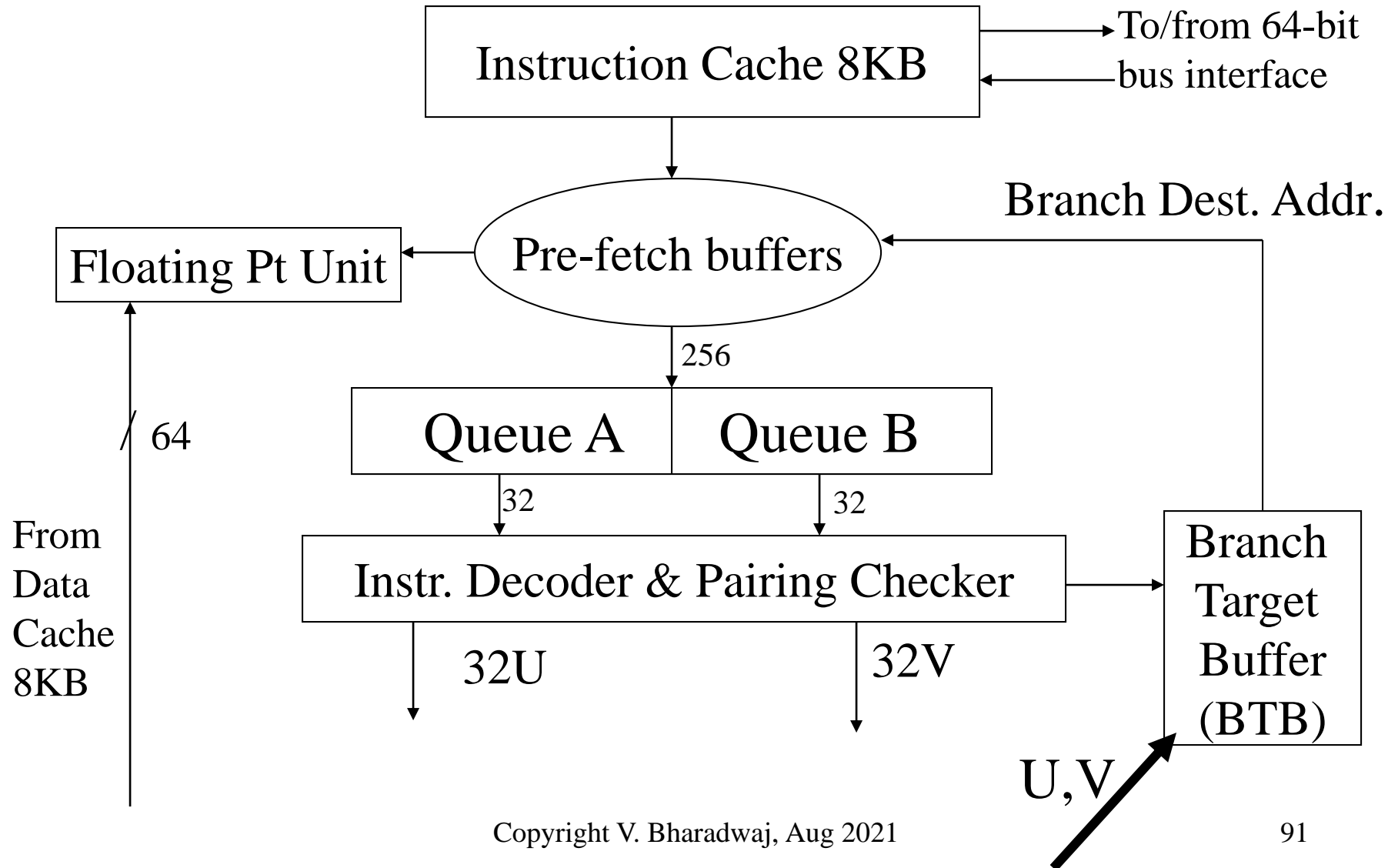
Latency minimization (CPU) with High Throughput (GPU) is achievable by this combo!

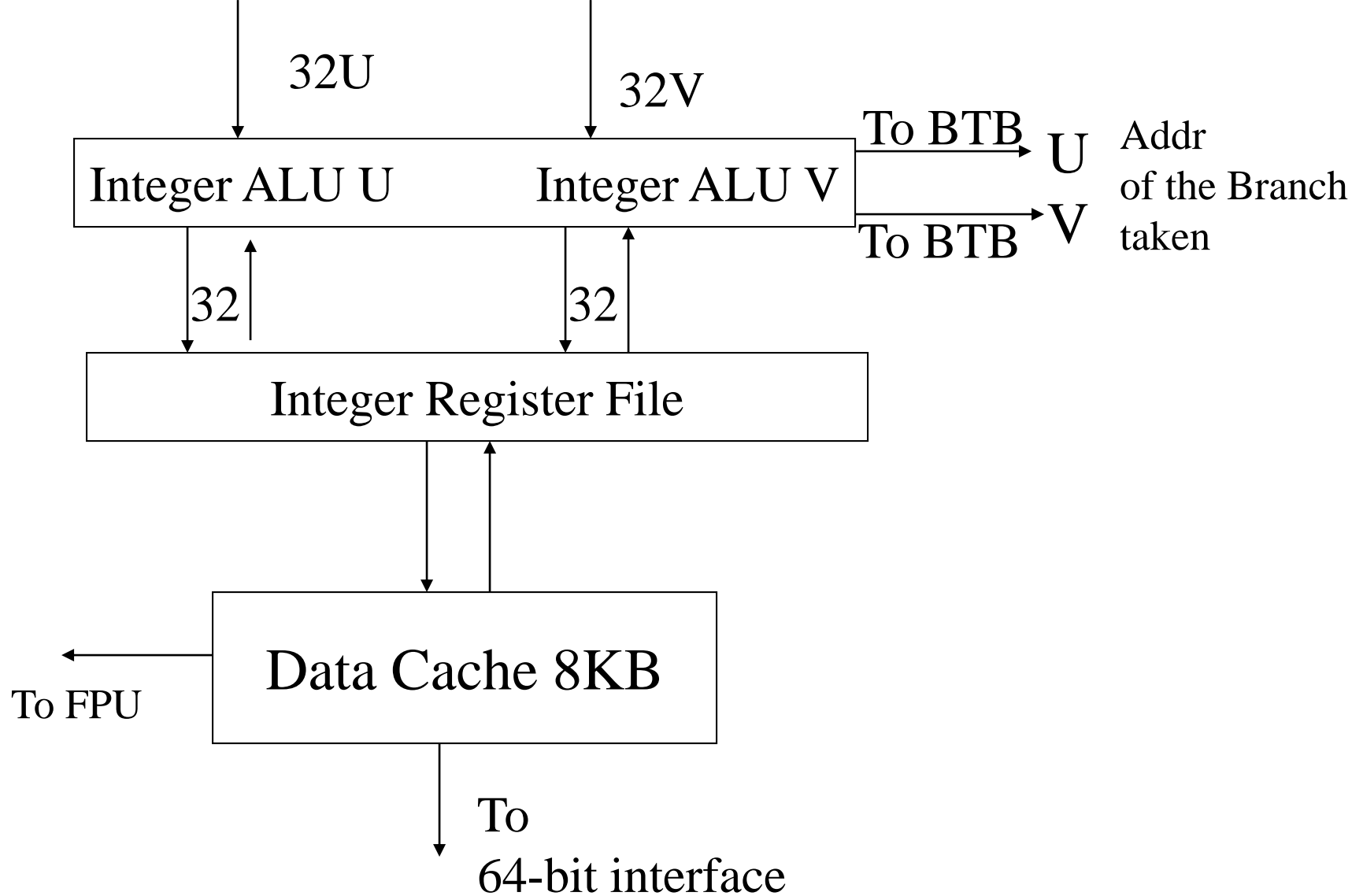
- How does a GPU architecture look like?
- What is the programming model?
- Example of a sample GPU program.

Refer to our web-page for details.

APPENDIX – Intel CPU Description of Pipeline action

(Pentium series ++)





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 .

**In-order
Front-end**

8-KB instruction cache

Simple decoder

1 micro-uop

Simple decoder

1 micro-uop

General decoder

4 micro-uops

Reorder
Buffer
(40 entries)

Micro-uop sequencer

Out-of-order RISC core

2 integer ALUS
2 load/store units
1 floating pt unit

System bus interface and level-2 cache interface

CPU

C
H
I
P

At the back-end, a superscalar execution engine is capable of executing *five μ ops 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 μ ops are first passed to a 40-entry reorder buffer shown, where they wait for the required operands become available.

From this reorder buffer, the μ ops 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

Appendix:

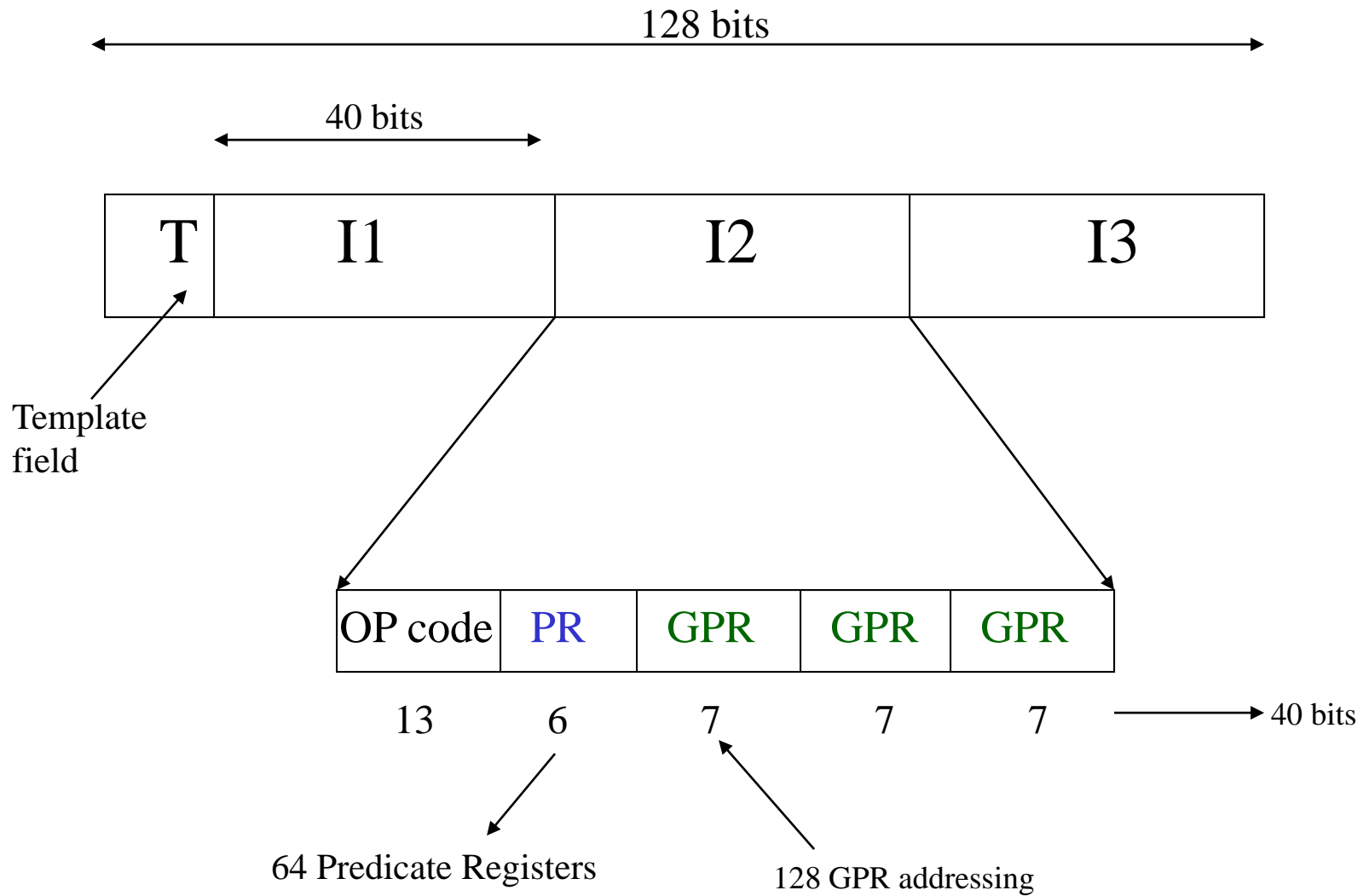
IA – 64 (Itanium) Processor Architecture (Classical)

Ref: Itanium Processor Micro-architecture, IEEE Micro, Sept-Oct 2000, Pp.24-43

Intel's effort with HP as the next generation architecture. It is called **Explicitly Parallel Instruction Computing (EPIC)** architecture – a big change from Pentium series. IA-64 – code named **Itanium**.

Uses a **128 bit Instruction word** comprising **3 instructions**; Each instruction within it has 40 bits in length -
3 GPR fields(7 bits each → 21 bits), 1 PR (predicate register 6bits);
13 bit opcode;


Innovation of IA-64 is in this PR register;



IA-64 Instruction Format

These registers are used to speculatively execute instructions across branches;

An intelligent compiler is behind the design – it detects explicitly which instructions (out of 3) can be carried out in parallel. This information is put in a *8-bit field (template field (part of the 128 bits)) which identifies the instructions in the current bundle (of 3 instructions) and in the following bundle (of 3 instructions) that can be carried out in parallel.*

Speculative loading  of data from memory is also carried out to match the CPU-memory speeds; Idea is to reorder instructions and issue a speculative look-ahead of a branch instruction.

IBM's Cell Processor Architecture (2007)

(non-examinable)

Refer to my write-up in our download zone of our Course page!

Other processor architectures – (non-examinable!!)

Sun's Niagara - <http://www.sun.com/processors/niagara/>

NVIDIA CUDA - <http://developer.nvidia.com/object/cuda.html>