MODULE-2

Chapter 4 Processors and Memory Hierarchy

4.1 Advanced Processor Technology

4.1.1 Design Space of Processors

- Processors can be "mapped" to a space that has clock rate and cycles per instruction (CPI) as coordinates. Each processor type occupies a region of this space.
- Newer technologies are enabling higher clock rates.
- Manufacturers are also trying to lower the number of cycles per instruction.
- Thus the "future processor space" is moving toward the lower right of the processor design space.

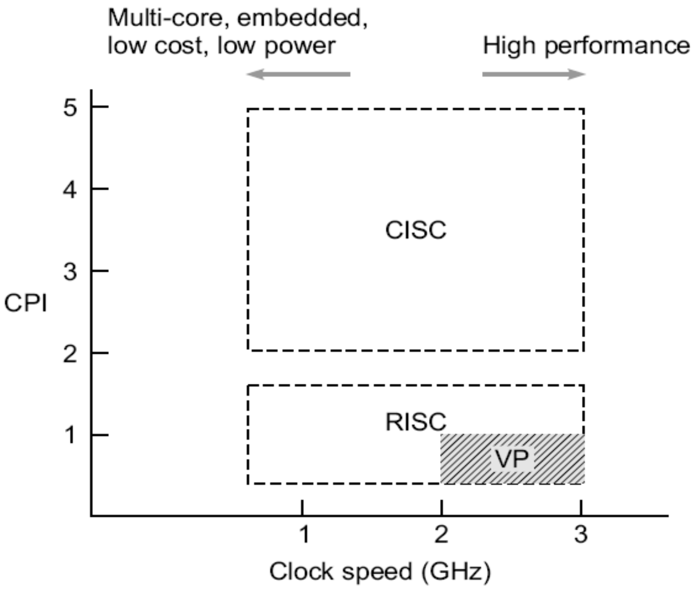


Fig. 4.1 CPI versus processor clock speed of major categories of processors

CISC and RISC Processors

- Complex Instruction Set Computing (CISC) processors like the Intel 80486, the Motorola 68040, the VAX/8600, and the IBM S/390 typically use microprogrammed control units, have lower clock rates, and higher CPI figures than…
- Reduced Instruction Set Computing (RISC) processors like the Intel i860, SPARC, MIPS R3000, and IBM RS/6000, which have hard-wired control units, higher clock rates, and lower CPI figures.

Superscalar Processors

- This subclass of the RISC processors allow multiple instructions to be issued simultaneously during each cycle.

- The effective CPI of a superscalar processor should be less than that of a generic scalar RISC processor.
- Clock rates of scalar RISC and superscalar RISC machines are similar.

VLIW Machines

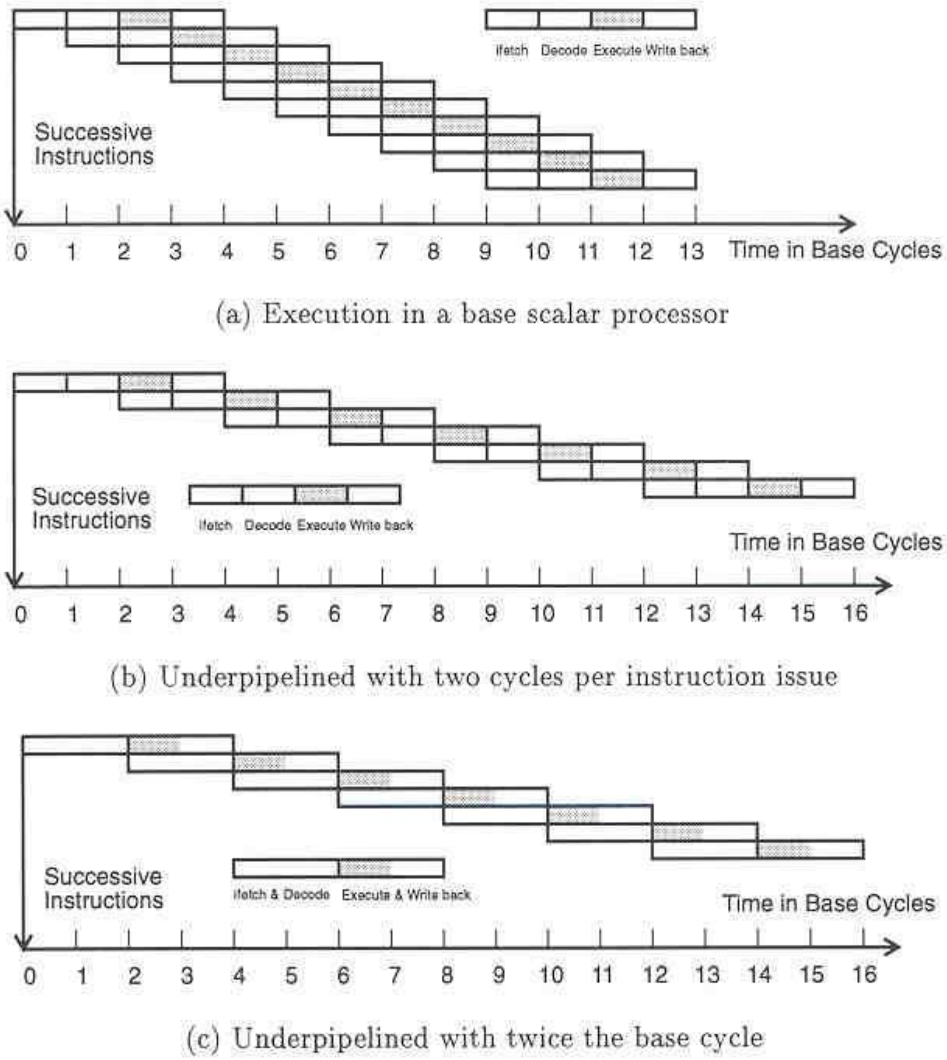
- Very Long Instruction Word machines typically have many more functional units than superscalars (and thus the need for longer – 256 to 1024 bits – instructions to provide control for them).
- These machines mostly use microprogrammed control units with relatively slow clock rates because of the need to use ROM to hold the microcode.

Superpipelined Processors

- These processors typically use a multiphase clock (actually several clocks that are out of phase with each other, each phase perhaps controlling the issue of another instruction) running at a relatively high rate.
- The CPI in these machines tends to be relatively high (unless multiple instruction issue is used).
- Processors in vector supercomputers are mostly superpipelined and use multiple functional units for concurrent scalar and vector operations.

Instruction Pipelines

- Typical instruction includes four phases:
  - fetch
  - decode
  - execute
  - write-back
- These four phases are frequently performed in a pipeline, or “assembly line” manner, as illustrated on the figure 4.2.
- The pipeline, like an industrial assembly line, receives successive instructions from its input end and executes them in a streamlined, overlapped fashion as they flow through.
- A pipeline cycle is intuitively defined as the time required for each phase to complete its operation, assuming equal delay in all phases (pipeline stages).



(a) Execution in a base scalar processor

(b) Underpipelined with two cycles per instruction issue

(c) Underpipelined with twice the base cycle

Figure 4.2 Pipelined execution of successive instructions in a base scalar processor and in two underpipelined cases. Courtesy of Jouppi and Wall; reprinted from *Proc. ASPLOS*, ACM Press, 1989)

Basic definitions associated with Pipeline operations:

- **Instruction pipeline cycle** – the time required for each phase to complete its operation (assuming equal delay in all phases)

- **Instruction issue latency** – the time (in cycles) required between the issuing of two adjacent instructions

- **Instruction issue rate** – the number of instructions issued per cycle (the *degree* of a superscalar)

- **Simple operation latency** – the delay (after the previous instruction) associated with the completion of a simple operation (e.g. integer add) as compared with that of a complex operation (e.g. divide).

- **Resource conflicts** – when two or more instructions demand use of the same functional unit(s) at the same time.

Pipelined Processors

- **A *base scalar processor***:
  - issues one instruction per cycle
  - has a one-cycle latency for a simple operation
  - has a one-cycle latency between instruction issues
  - can be fully utilized if instructions can enter the pipeline at a rate on one per cycle
- For a variety of reasons, instructions might not be able to be pipelines as aggressively as in a base scalar processor. In these cases, we say the pipeline is *underpipelined*.
- CPI rating is 1 for an ideal pipeline. Underpipelined systems will have higher CPI ratings, lower clock rates, or both.

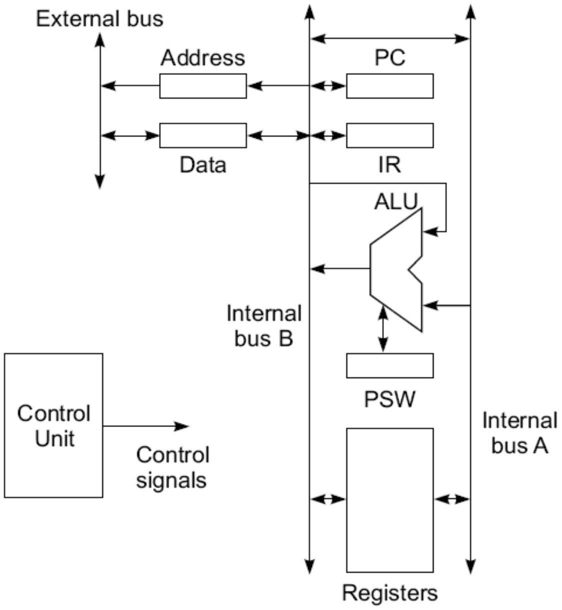


Fig. 4.3 Data path architecture and control unit of a scalar processor

- Figure 4.3 shows the data path architecture and control unit of a typical, simple scalar processor which does not employ an instruction pipeline. Main memory, I/O controllers, etc. are connected to the external bus.
- The control unit generates control signals required for the *fetch*, *decode*, *ALU operation*, *memory access*, and *write result* phases of instruction execution.

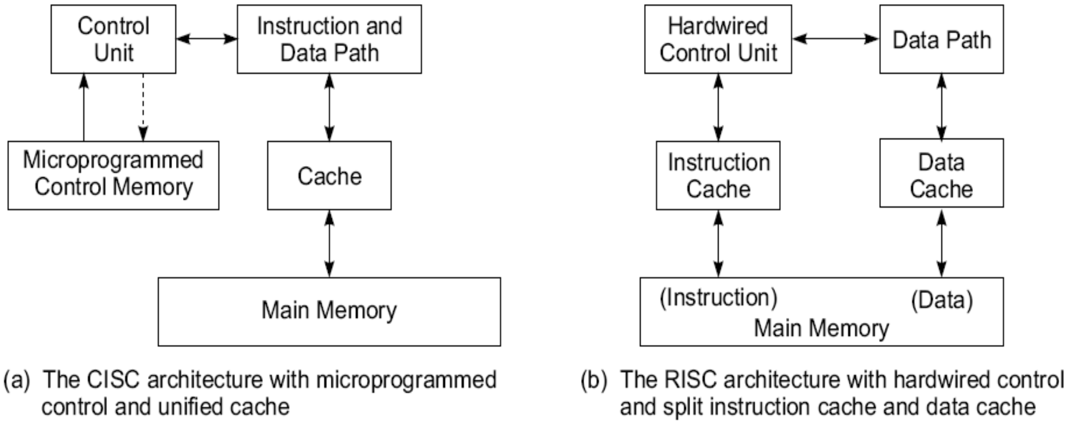
- The control unit itself may employ hardwired logic, or—as was more common in older CISC style processors—microcoded logic.
- Modern RISC processors employ hardwired logic, and even modern CISC processors make use of many of the techniques originally developed for high-performance RISC processors.

4.1.2 Instruction Set Architectures

- **CISC**
  - Many different instructions
  - Many different operand data types
  - Many different operand addressing formats
  - Relatively small number of general purpose registers
  - Many instructions directly match high-level language constructions
- **RISC**
  - Many fewer instructions than CISC (freeing chip space for more functional units!)
  - Fixed instruction format (e.g. 32 bits) and simple operand addressing
  - Relatively large number of registers
  - Small CPI (close to 1) and high clock rates

Architectural Distinctions

- **CISC**
  - Unified cache for instructions and data (in most cases)
  - Microprogrammed control units and ROM in earlier processors (hard-wired controls units now in some CISC systems)
- **RISC**
  - Separate instruction and data caches
  - Hard-wired control units

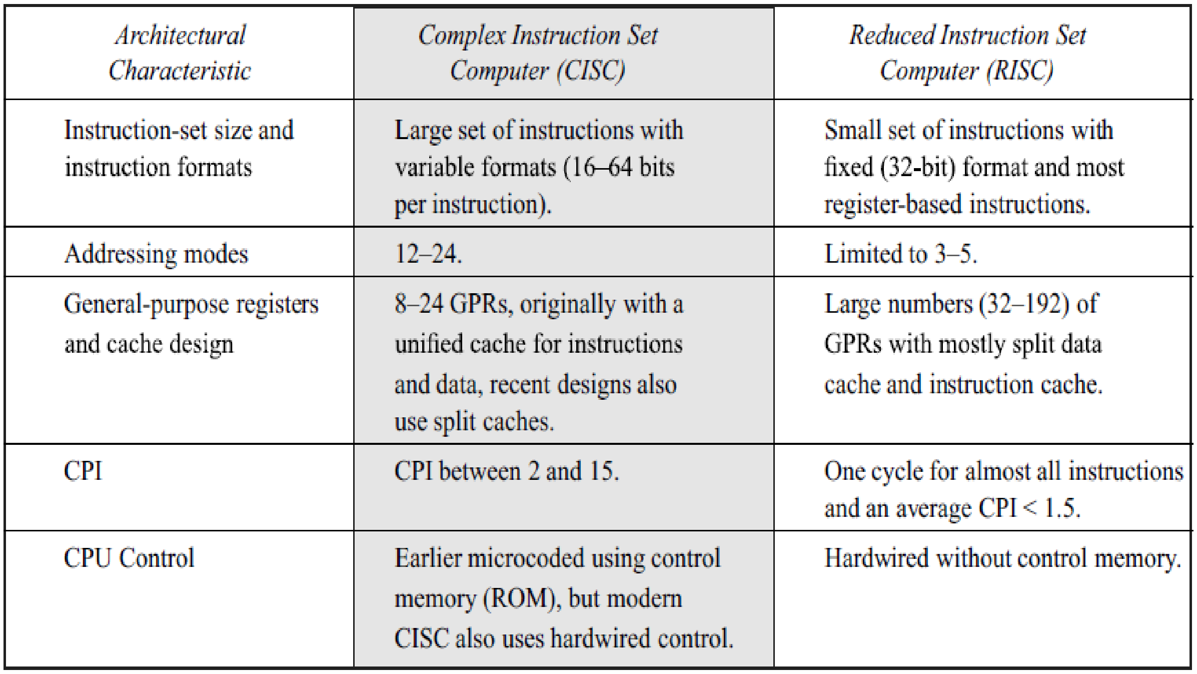


(a) The CISC architecture with microprogrammed control and unified cache

(b) The RISC architecture with hardwired control and split instruction cache and data cache

Fig. 4.4 Distinctions between typical RISC and typical CISC processor architectures (Courtesy of Gordon Bell, 1989)

Table 4.1 *Characteristics of Typical CISC and RISC Architectures*



Architectural Characteristic	*Complex Instruction Set Computer (CISC)*	*Reduced Instruction Set Computer (RISC)*
Instruction-set size and instruction formats	Large set of instructions with variable formats (16–64 bits per instruction).	Small set of instructions with fixed (32-bit) format and most register-based instructions.
Addressing modes	12–24.	Limited to 3–5.
General-purpose registers and cache design	8–24 GPRs, originally with a unified cache for instructions and data, recent designs also use split caches.	Large numbers (32–192) of GPRs with mostly split data cache and instruction cache.
CPI	CPI between 2 and 15.	One cycle for almost all instructions and an average CPI < 1.5.
CPU Control	Earlier microcoded using control memory (ROM), but modern CISC also uses hardwired control.	Hardwired without control memory.

- **CISC Advantages**
  - Smaller program size (fewer instructions)
  - Simpler control unit design
  - Simpler compiler design

- **RISC Advantages**
  - Has potential to be faster
  - Many more registers
- **RISC Problems**
  - More complicated register decoding system
  - Hardwired control is less flexible than microcode

4.1.3 CISC Scalar Processors

- Early systems had only integer fixed point facilities.
- Modern machines have both fixed and floating point facilities, sometimes as parallel functional units.
- Many CISC scalar machines are underpipelined.

Representative CISC Processors:

- VAX 8600
- Motorola MC68040
- Intel Pentium

VAX 8600 processor

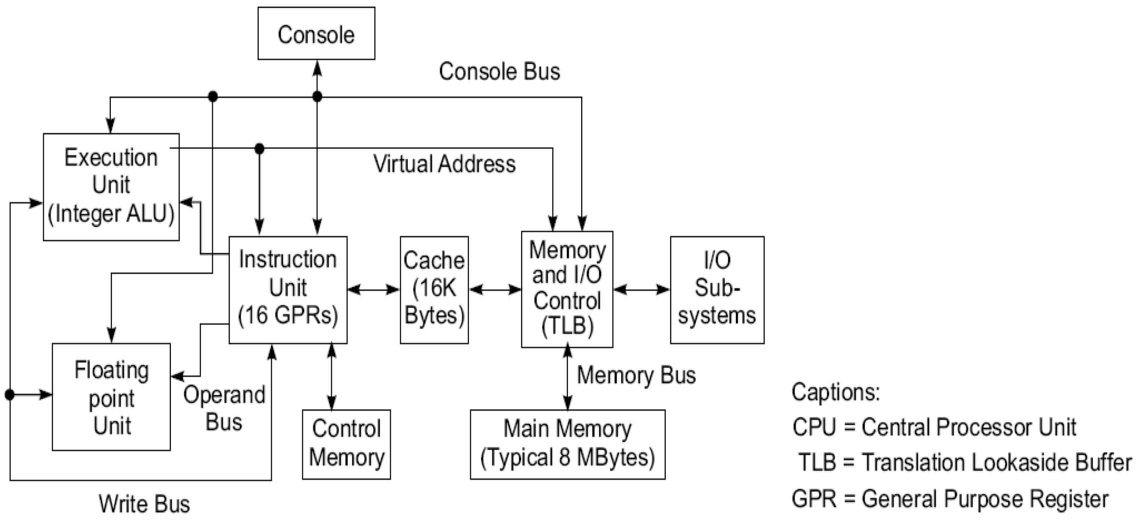

Console
Console Bus
Execution Unit (Integer ALU)
Virtual Address
Instruction Unit (16 GPRs)
Cache (16K Bytes)
Memory and I/O Control (TLB)
I/O Sub-systems
Floating point Unit
Operand Bus
Control Memory
Memory Bus
Main Memory (Typical 8 MBytes)
Write Bus
Captions:
CPU = Central Processor Unit
TLB = Translation Lookaside Buffer
GPR = General Purpose Register

Fig. 4.5 The VAX 8600 CPU, a typical CISC processor architecture (Courtesy of Digital Equipment Corporation, 1985)

- The VAX 8600 was introduced by Digital Equipment Corporation in 1985.
- This machine implemented a typical CISC architecture with microprogrammed control.
- The instruction set contained about 300 instructions with 20 different addressing modes.
- The CPU in the VAX 8600 consisted of two functional units for concurrent execution of integer and floating point instructions.
- The unified cache was used for holding both instructions and data.
- There were 16 GPRs in the instruction unit. Instruction pipelining was built with six stages in the VAX 8600, as in most elsc machines.
- The instruction unit prefetched and decoded instructions, handled branching operations, and supplied operands to the two functional units in a pipelined fashion.
- A Translation Lookaside Buffer (TLB) was used in the memory control unit for fast generation of a physical address from a virtual address.
- Both integer and floating point units were pipelined.
- The performance of the processor pipelines relied heavily on the cache hit ratio and on minimal branching damage to the pipeline flow.

4.1.4 RISC Scalar Processors

- Designed to issue one instruction per cycle
- RISC and CISC scalar processors should have same performance if clock rate and program lengths are equal.

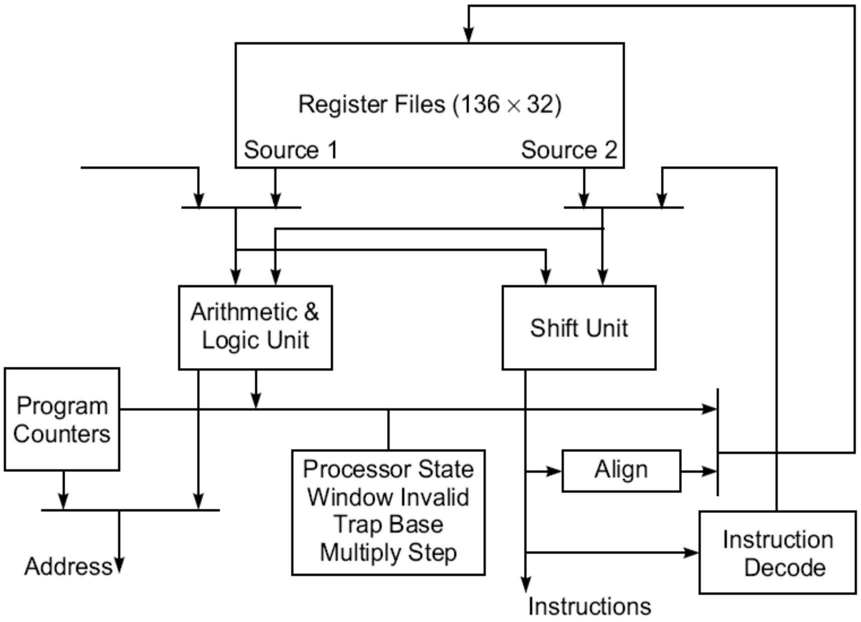
- RISC moves less frequent operations into software, thus dedicating hardware resources to the most frequently used operations.

Representative RISC Processors:

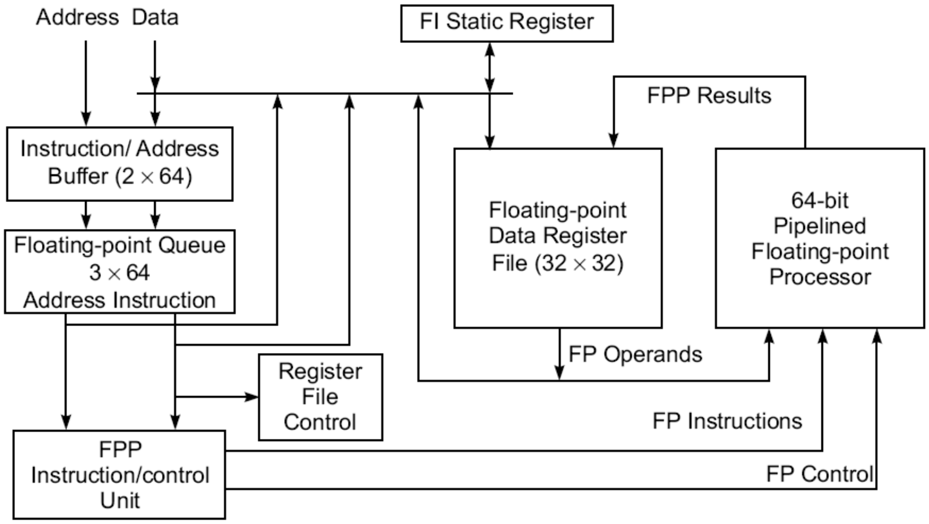
- Sun SPARC
- Intel i860
- Motorola M88100
- AMD 29000

SPARCs (Scalable Processor Architecture) and Register Windows

- SPARC family chips produced by Cypress Semiconductors, Inc. Figure 4.7 shows the architecture of the Cypress CY7C601 SPARC processor and of the CY7C602 FPU.
- The Sun SPARC instruction set contains 69 basic instructions
- The SPARC runs each procedure with a set of thirty-two 32-bit IU registers.
- Eight of these registers are ***global registers*** shared by all procedures, and the remaining 24 are ***window registers*** associated with only each procedure.
- The concept of using overlapped register windows is the most important feature introduced by the Berkeley RISC architecture.



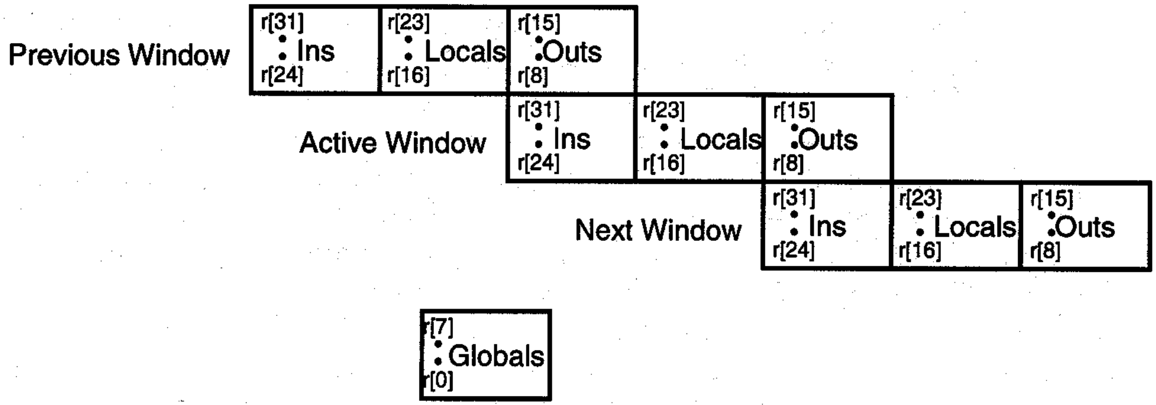
(a) The Cypress CY7C601 SPARC processor


Address
Data
FI Static Register
FPP Results
Instruction/ Address Buffer (2 × 64)
Floating-point Queue 3 × 64 Address Instruction
Floating-point Data Register File (32 × 32)
64-bit Pipelined Floating-point Processor
FP Operands
Register File Control
FP Instructions
FPP Instruction/control Unit
FP Control

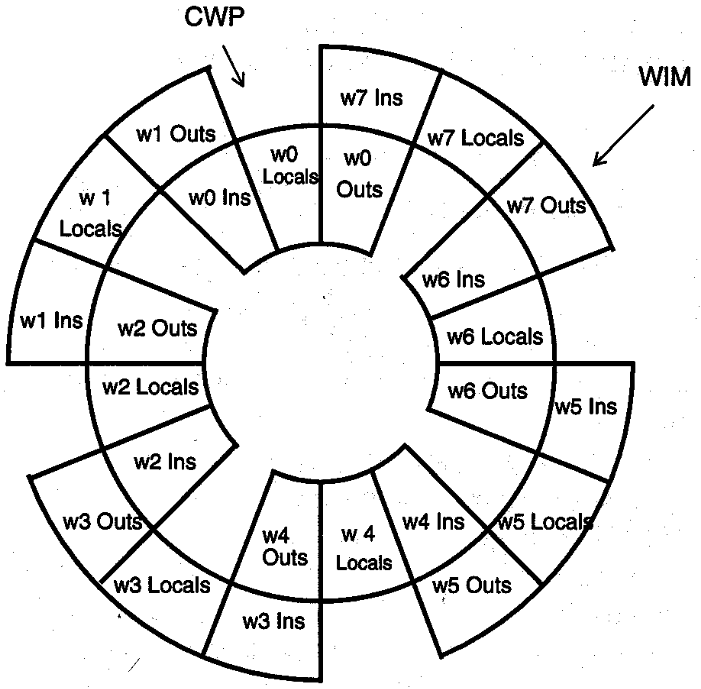
(b) The Cypress CY7C602 floating-point unit

Fig. 4.7 The SPARC architecture with the processor and the floating-point unit on two separate chips (Courtesy of Cypress Semiconductor Co., 1991)

- Fig. 4.8 shows eight overlapping windows (formed with 64 **local registers** and 64 overlapped registers) and eight **globals** with a total of 136 registers, as implemented in the Cypress 601.
- Each register window is divided into three eight-register sections, labeled ***Ins***, ***Locals***, and ***Outs***.
- The local registers are only locally addressable by each procedure. The Ins and Outs are shared among procedures.
- The calling procedure passes parameters to the called procedure via its Outs (r8 to r15) registers, which are the Ins registers of the called procedure.
- The window of the currently running procedure is called the active window pointed to by a current window pointer.
- A window invalid mask is used to indicate which window is invalid. The trap base register serves as a pointer to a trap handler.



(a) Three overlapping register windows and the global registers



(b) Eight register windows forming a circular stack

Figure 4.8 The concept of overlapping register windows in the SPARC architecture. (Courtesy of Sun Microsystems, Inc., 1987)

- A special register is used to create a 64-bit product in multiple step instructions. Procedures can also be called without changing the window.
- The overlapping windows can significantly save the time required for interprocedure communications, resulting in much faster context switching among cooperative procedures.

4.2 Superscalar, Vector Processors

- A CISC or a RISC scalar processor can be improved with a superscalar or vector architecture.
- Scalar processors are those executing one instruction per cycle.
- Only one instruction is issued per cycle, and only one completion of instruction is expected from the pipeline per cycle.
- In a superscalar processor, multiple instructions are issued per cycle and multiple results are generated per cycle.
- A vector processor executes vector instructions on arrays of data; each vector instruction involves a string of repeated operations, which are ideal for pipelining with one result per cycle.

4.2.1 Superscalar Processors

- Superscalar processors are designed to exploit more instruction-level parallelism in user programs.
- Only independent instructions can be executed in parallel without causing a wait state. The amount of instruction level parallelism varies widely depending on the type of code being executed.
- It has been observed that the average value is around 2 for code without loop unrolling. Therefore, for these codes there is not much benefit gained from building a machine that can issue more than three instructions per cycle.
- The instruction-issue degree in a superscalar processor has thus been limited to 2 to 5 in practice.

Pipelining in Superscalar Processors

- The fundamental structure of a three-issue superscalar pipeline is illustrated in Fig. 4.11.
- Superscalar processors were originally developed as an alternative to vector processors, with a view to exploit higher degree of instruction level parallelism.

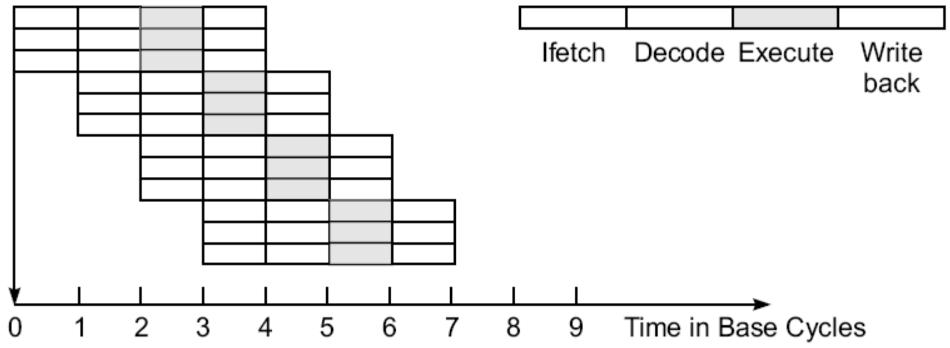


Fig. 4.11 A superscalar processor of degree m = 3

- A superscalar processor of degree m can issue m instructions per cycle.
- The base scalar processor, implemented either in RISC or CISC, has m = 1.
- In order to fully utilize a superscalar processor of degree m, m instructions must be executable in parallel. This situation may not be true in all clock cycles.
- In that case, some of the pipelines may be stalling in a wait state.
- In a superscalar processor, the simple operation latency should require only one cycle, as in the base scalar processor.
- Due to the desire for a higher degree of instruction-level parallelism in programs, the superscalar processor depends more on an optimizing compiler to exploit parallelism.

Representative Superscalar Processors

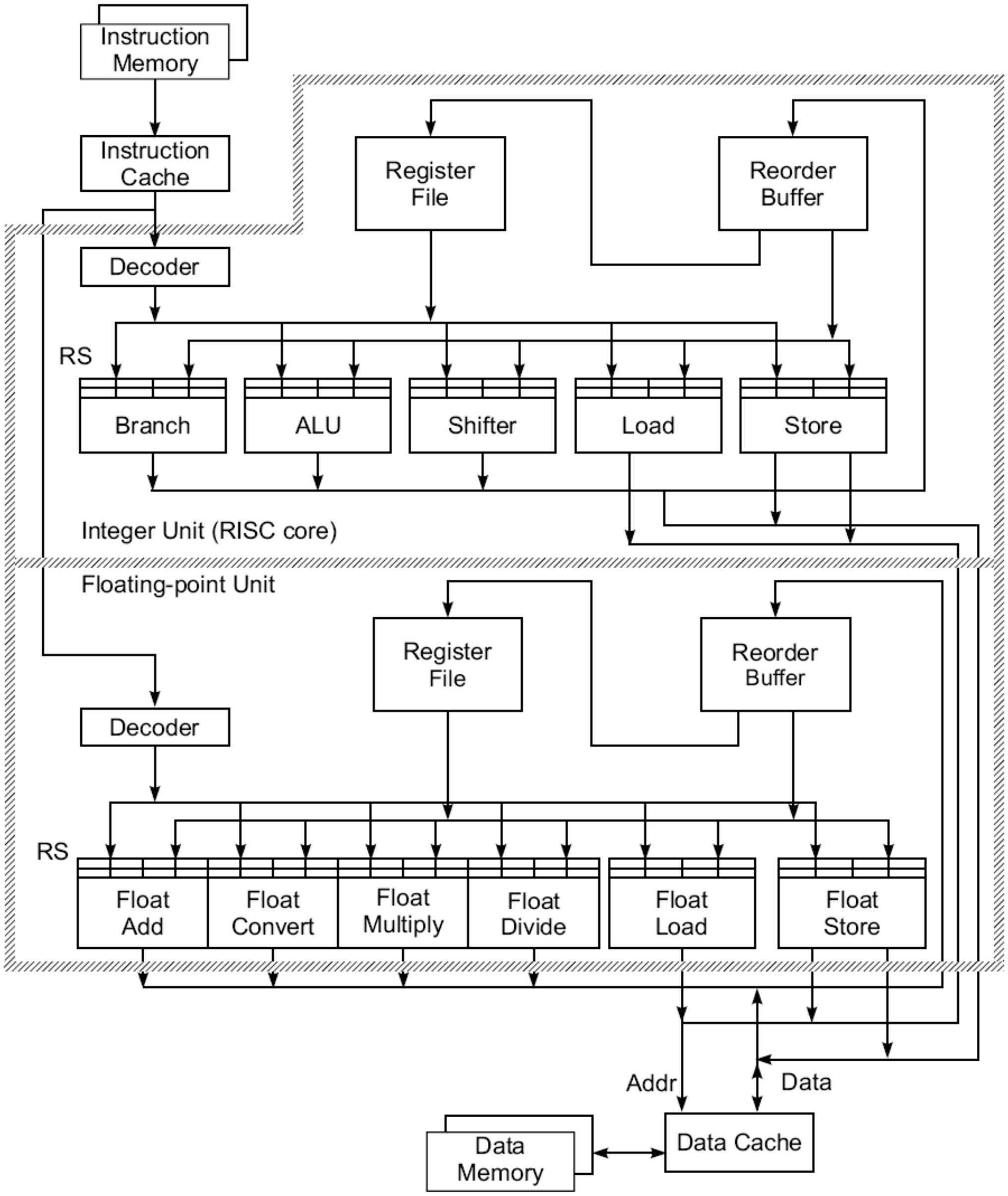


Fig. 4.12 A typical superscalar RISC processor architecture consisting of an integer unit and a floating-point unit (Courtesy of M. Johnson, 1991; reprinted with permission from Prentice-Hall, Inc.)

Typical Superscalar Architecture

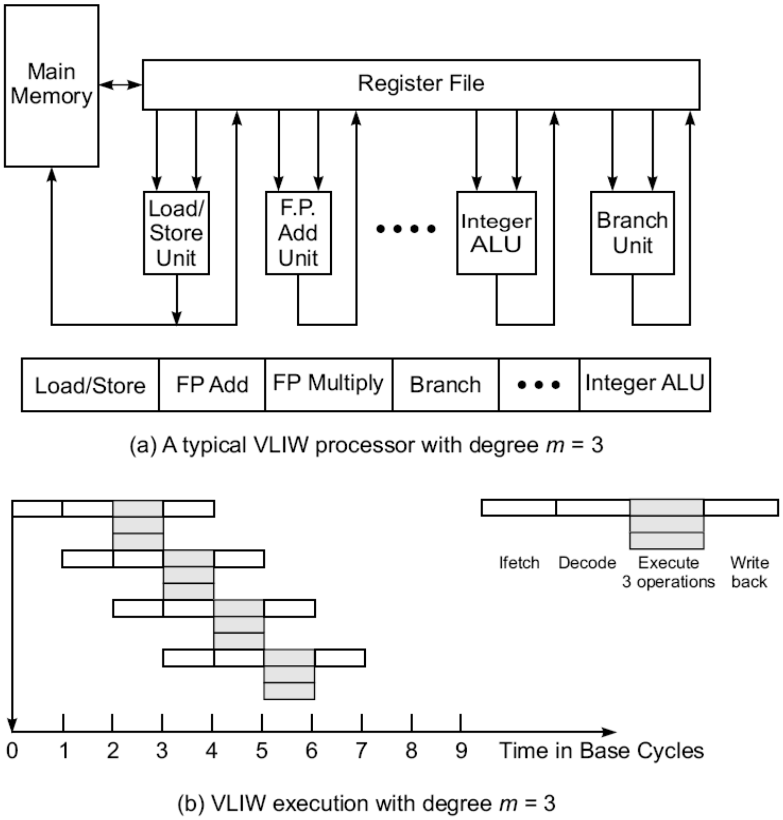
- A typical superscalar will have
  - multiple instruction pipelines
  - an instruction cache that can provide multiple instructions per fetch
  - multiple buses among the function units
- In theory, all functional units can be simultaneously active.

4.2.2 VLIW Architecture

- VLIW = Very Long Instruction Word
- Instructions usually hundreds of bits long.
- Each instruction word essentially carries multiple "short instructions."
- Each of the "short instructions" are effectively issued at the same time.
- (This is related to the long words frequently used in microcode.)
- Compilers for VLIW architectures should optimally try to predict branch outcomes to properly group instructions.

Pipelining in VLIW Processors

- Decoding of instructions is easier in VLIW than in superscalars, because each "region" of an instruction word is usually limited as to the type of instruction it can contain.
- Code density in VLIW is less than in superscalars, because if a "region" of a VLIW word isn't needed in a particular instruction, it must still exist (to be filled with a "no op").
- Superscalars can be compatible with scalar processors; this is difficult with VLIW parallel and non-parallel architectures.



(a) A typical VLIW processor with degree $m = 3$

(b) VLIW execution with degree $m = 3$

Fig. 4.14 The architecture of a very long instruction word (VLIW) processor and its pipeline operations (Courtesy of Multiflow Computer, Inc., 1987)

VLIW Opportunities

- “Random” parallelism among scalar operations is exploited in VLIW, instead of regular parallelism in a vector or SIMD machine.
- The efficiency of the machine is entirely dictated by the success, or “goodness,” of the compiler in planning the operations to be placed in the same instruction words.
- Different implementations of the same VLIW architecture may not be binary-compatible with each other, resulting in different latencies.

VLIW Summary

- VLIW reduces the effort required to detect parallelism using hardware or software techniques.
- The main advantage of VLIW architecture is its simplicity in hardware structure and instruction set.
- Unfortunately, VLIW does require careful analysis of code in order to “compact” the most appropriate ”short” instructions into a VLIW word.

4.2.3 Vector Processors

- A vector processor is a coprocessor designed to perform vector computations.
- A vector is a one-dimensional array of data items (each of the same data type).
- Vector processors are often used in multipipelined supercomputers.

Architectural types include:

1. **Register-to-Register** (with shorter instructions and register files)
2. **Memory-to-Memory** (longer instructions with memory addresses)

1. Register-to-Register Vector Instructions

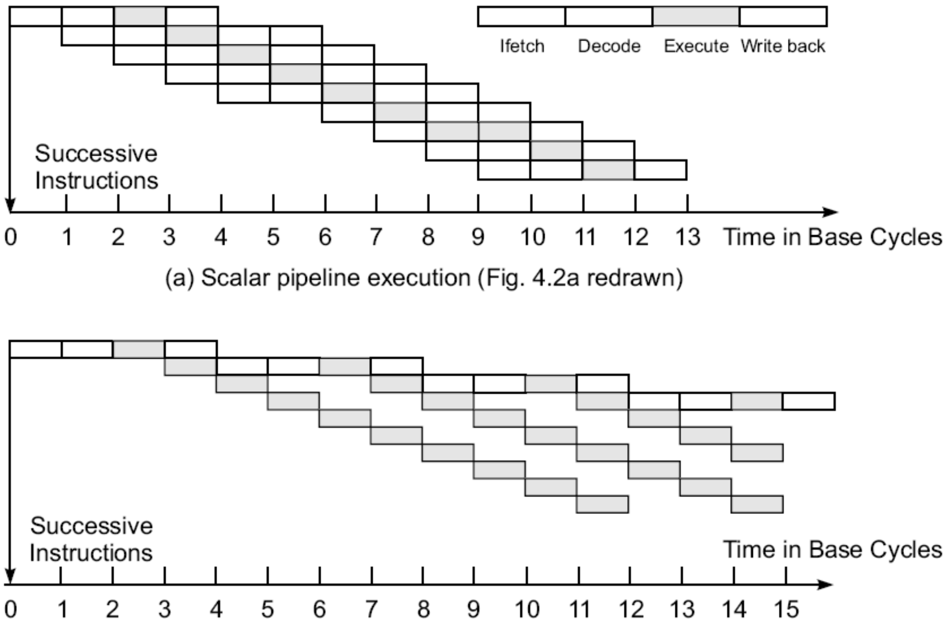
- Assume V_i is a vector register of length n, s_i is a scalar register, M(1:n) is a memory array of length n, and “o” is a vector operation.
- Typical instructions include the following:
  - $\mathbf{V_1 \text{ o } V_2 \rightarrow V_3}$ (element by element operation)
  - $\mathbf{s_1 \text{ o } V_1 \rightarrow V_2}$ (scaling of each element)
  - $\mathbf{V_1 \text{ o } V_2 \rightarrow s_1}$ (binary reduction - i.e. sum of products)
  - $\mathbf{M(1:n) \rightarrow V_1}$ (load a vector register from memory)
  - $\mathbf{V_1 \rightarrow M(1:n)}$ (store a vector register into memory)

- **$o\ V_1 \rightarrow V_2$** (unary vector -- i.e. negation)
- **$o\ V_1 \rightarrow s_1$** (unary reduction -- i.e. sum of vector)

2. Memory-to-Memory Vector Instructions

- Typical memory-to-memory vector instructions (using the same notation as given in the previous slide) include these:
  - **$M_1(1{:}n)\ o\ M_2(1{:}n) \rightarrow M_3(1{:}\ n)$** (binary vector)
  - **$s_1\ o\ M_1(1{:}n) \rightarrow M_2(1{:}n)$** (scaling)
  - **$o\ M_1(1{:}n) \rightarrow M_2(1{:}n)$** (unary vector)
  - **$M_1(1{:}n)\ o\ M_2(1{:}n) \rightarrow M(k)$** (binary reduction)

Pipelines in Vector Processors



(a) Scalar pipeline execution (Fig. 4.2a redrawn)

(b) Vector pipeline execution

Fig. 4.15 Pipelined execution in a base scalar processor and in a vector processor, respectively (Courtesy of Jouppi and Wall; reprinted from Proc. ASPLOS, ACM Press, 1989)

- Vector processors can usually effectively use large pipelines in parallel, the number of such parallel pipelines effectively limited by the number of functional units.
- As usual, the effectiveness of a pipelined system depends on the availability and use of an effective compiler to generate code that makes good use of the pipeline facilities.

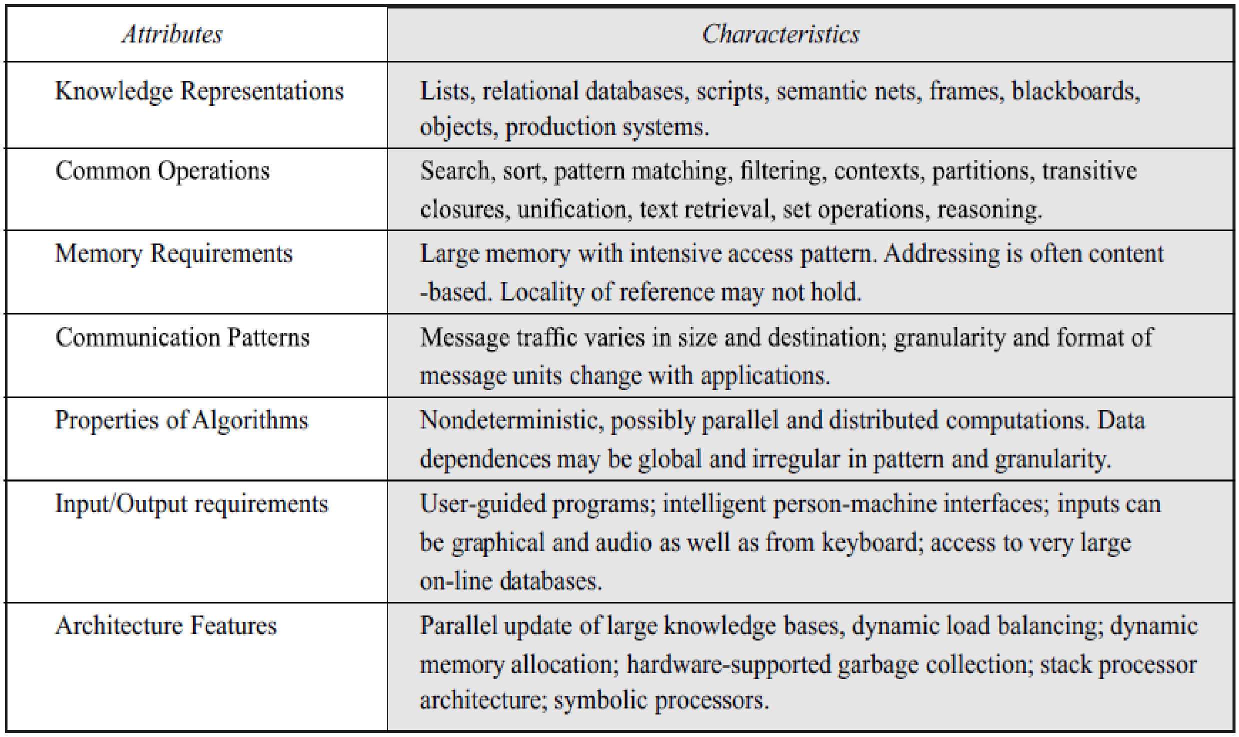
Symbolic Processors

- Symbolic processors are somewhat unique in that their architectures are tailored toward the execution of programs in languages similar to LISP, Scheme, and Prolog.

- In effect, the hardware provides a facility for the manipulation of the relevant data objects with "tailored" instructions.

- These processors (and programs of these types) may invalidate assumptions made about more traditional scientific and business computations.

Table 4.6 *Characteristics of Symbolic Processing*



Attributes	*Characteristics*
Knowledge Representations	Lists, relational databases, scripts, semantic nets, frames, blackboards, objects, production systems.
Common Operations	Search, sort, pattern matching, filtering, contexts, partitions, transitive closures, unification, text retrieval, set operations, reasoning.
Memory Requirements	Large memory with intensive access pattern. Addressing is often content -based. Locality of reference may not hold.
Communication Patterns	Message traffic varies in size and destination; granularity and format of message units change with applications.
Properties of Algorithms	Nondeterministic, possibly parallel and distributed computations. Data dependences may be global and irregular in pattern and granularity.
Input/Output requirements	User-guided programs; intelligent person-machine interfaces; inputs can be graphical and audio as well as from keyboard; access to very large on-line databases.
Architecture Features	Parallel update of large knowledge bases, dynamic load balancing; dynamic memory allocation; hardware-supported garbage collection; stack processor architecture; symbolic processors.

4.3 Memory Hierarchical Technology

- Storage devices such as registers, caches, main memory, disk devices, and backup storage are often organized as a hierarchy as depicted in Fig. 4.17.
- The memory technology and storage organization at each level is characterized by five parameters:

  1. **access time t_i** (round-trip time from CPU to ith level)
  2. **memory size s_i** (number of bytes or words in level i)
  3. **cost per byte c_i**
  4. **transfer bandwidth b_i** (rate of transfer between levels)
  5. **unit of transfer x_i** (grain size for transfers between levels i and i+1)

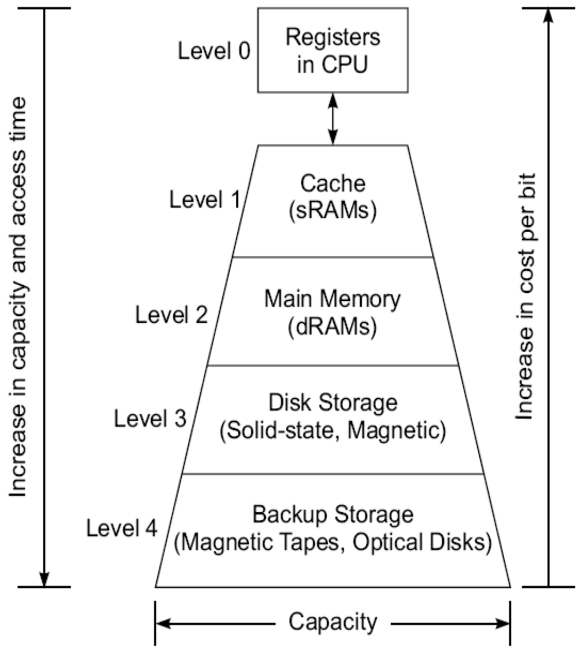

Level 0
Registers in CPU
Level 1
Cache (sRAMs)
Level 2
Main Memory (dRAMs)
Level 3
Disk Storage (Solid-state, Magnetic)
Level 4
Backup Storage (Magnetic Tapes, Optical Disks)
Capacity
Increase in capacity and access time
Increase in cost per bit

Fig. 4.17 A four-level memory hierarchy with increasing capacity and decreasing speed and cost from low to high levels

Memory devices at a lower level are:

- faster to access,
- are smaller in capacity,
- are more expensive per byte,
- have a higher bandwidth, and
- have a smaller unit of transfer.

In general, $\mathbf{t_{i-1} < t_i}$, $\mathbf{s_{i-1} < s_i}$, $\mathbf{c_{i-1} > c_i}$, $\mathbf{b_{i-1} > b_i}$ **and** $\mathbf{x_{i-1} < x_i}$ for i = 1, 2, 3, and 4 in the hierarchy where i = 0 corresponds to the CPU register level.

The cache is at level 1, main memory at level 2, the disks at level 3 and backup storage at level 4.

Registers and Caches

Registers

- The registers are parts of the processor;
- Register assignment is made by the compiler.
- Register transfer operations are directly controlled by the processor after instructions are decoded.
- Register transfer is conducted at processor speed, in one clock cycle.

Caches

- The cache is controlled by the MMU and is programmer-transparent.
- The cache can also be implemented at one or multiple levels, depending on the speed and application requirements.
- Multi-level caches are built either on the processor chip or on the processor board.
- Multi-level cache systems have become essential to deal with memory access latency.

Main Memory (Primary Memory)

- It is usually much larger than the cache and often implemented by the most cost-effective RAM chips, such as DDR SDRAMs, i.e. dual data rate synchronous dynamic RAMs.
- The main memory is managed by a MMU in cooperation with the operating system.

Disk Drives and Backup Storage

- The disk storage is considered the highest level of on-line memory.
- It holds the system programs such as the OS and compilers, and user programs and their data sets.
- Optical disks and magnetic tape units are off-line memory for use as archival and backup storage.
- They hold copies of present and past user programs and processed results and files.
- Disk drives are also available in the form of RAID arrays.

Peripheral Technology

- Peripheral devices include printers, plotters, terminals, monitors, graphics displays, optical scanners, image digitizers, output microfilm devices etc.
- Some I/O devices are tied to special-purpose or multimedia applications.

4.3.2 Inclusion, Coherence, and Locality

Information stored in a memory hierarchy (M1, M2,…, Mn) satisfies 3 important properties:

1. **Inclusion**
2. **Coherence**
3. **Locality**

- We consider cache memory the innermost level M1, which directly communicates with the CPU registers.
- The outermost level M_n contains all the information words stored. In fact, the collection of all addressable words in M_n forms the virtual address space of a computer.

- Program and data locality is characterized below as the foundation for using a memory hierarchy effectively.

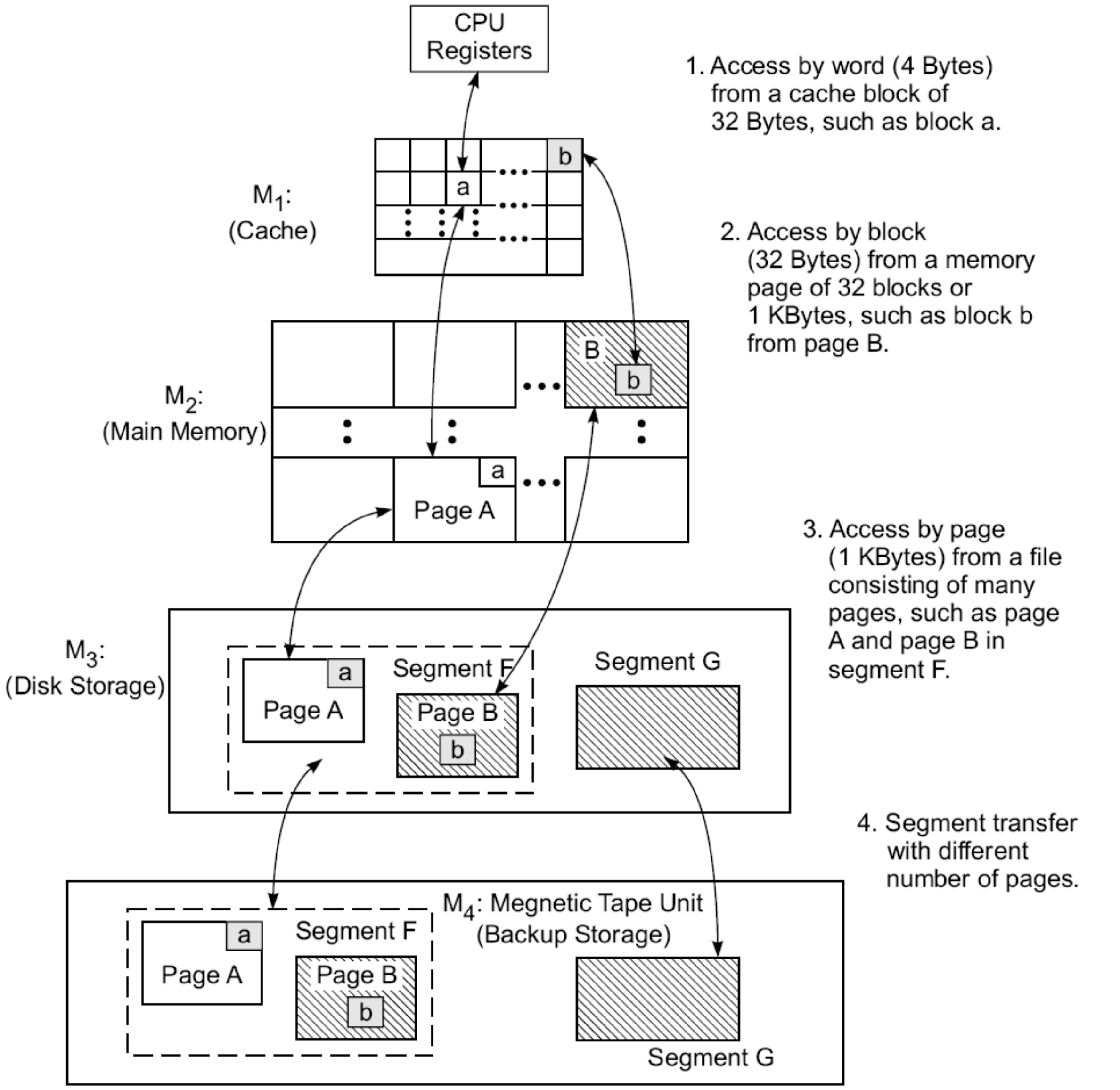


Fig. 4.18 The inclusion property and data transfers between adjacent levels of a memory hierarchy

1. The Inclusion Property

- The inclusion property is stated as:

  $\mathbf{M_1 \subset M_2 \subset ... \subset M_n}$

- The implication of the inclusion property is that all items of information in the "innermost" memory level (cache) also appear in the outer memory levels.

- The inverse, however, is not necessarily true. That is, the presence of a data item in level M_{i+1} does not imply its presence in level M_i. We call a reference to a missing item a "miss."

2. The Coherence Property

The requirement that copies of data items at successive memory levels be **consistent** is called the "coherence property."

Coherence Strategies

- **Write-through**
  - As soon as a data item in M_i is modified, immediate update of the corresponding data item(s) in M_{i+1}, M_{i+2}, … M_n is required.
  - This is the most aggressive (and expensive) strategy.
- **Write-back**
  - The update of the data item in M_{i+1} corresponding to a modified item in M_i is not updated unit it (or the block/page/etc. in M_i that contains it) is replaced or removed.
  - This is the most efficient approach, but cannot be used (without modification) when multiple processors share M_{i+1}, …, M_n.

3. Locality of References

- Memory references are generated by the CPU for either instruction or data access.
- These accesses tend to be clustered in certain regions in time, space, and ordering.

  There are three dimensions of the locality property:

  - ***Temporal locality*** – if location M is referenced at time t, then it (location M) will be referenced again at some time **t+Δt**.
  - ***Spatial locality*** – if location M is referenced at time t, then another location M±Δm will be referenced at time **t+Δt**.
  - ***Sequential locality*** – if location **M** is referenced at time **t**, then locations **M+1, M+2, …** will be referenced at time **t+Δt**, **t+Δt'**, etc.

- In each of these patterns, both Δm and Δt are "small."
- H&P suggest that 90 percent of the execution time in most programs is spent executing only 10 percent of the code.

Working Sets

- The set of addresses (bytes, pages, etc.) referenced by a program during the interval from t to t+ Δt, where Δt is called the *working set parameter*, changes slowly.
- This set of addresses, called the *working set*, should be present in the higher levels of M if a program is to execute efficiently (that is, without requiring numerous movements of data items from lower levels of M). This is called the *working set principle*.

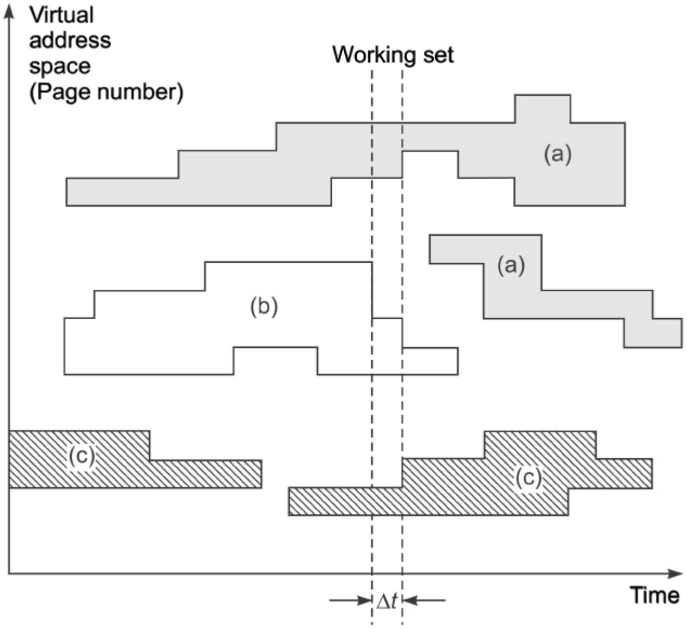


Fig. 4.19 Memory reference patterns in typical program trace experiments, where regions (a), (b), and (c) are generated with the execution of three software processes

4.3.3 Memory Capacity Planning

The performance of a memory hierarchy is determined by the Effective Access Time $\mathbf{T_{eff}}$ to any level in the hierarchy. It depends on the hit ratios and access frequencies.

Hit Ratios

- When a needed item (instruction or data) is found in the level of the memory hierarchy being examined, it is called a *hit*. Otherwise (when it is not found), it is called a *miss* (and the item must be obtained from a lower level in the hierarchy).
- The *hit ratio*, *h*, for M_i is the probability (between 0 and 1) that a needed data item is found when sought in level memory M_i.
- The *miss ratio* is obviously just $1-h_i$.

- We assume $h_0 = 0$ and $h_n = 1$.

Access Frequencies

- The access frequency f_i to level M_i is

  $$\mathbf{f_i = (1-h_1) \times (1-h_2) \times \ldots \times h_i}$$

- Note that $f_1 = h_1$, and $\sum_{i=1}^{n} \mathbf{fi} = 1$

Effective Access Times

- There are different penalties associated with misses at different levels in the memory hierarcy.
  - A cache miss is typically 2 to 4 times as expensive as a cache hit (assuming success at the next level).
  - A page fault (miss) is 3 to 4 magnitudes as costly as a page hit.
- The effective access time of a memory hierarchy can be expressed as

$$T_{eff} = \sum_{i=1}^{n} f_i \cdot t_i$$

$$= h_1 t_1 + (1-h_1) h_2 t_2 + \cdots + (1-h_1)(1-h_2) \cdots (1-h_{n-1}) h_n t_n$$

- The effective access time is still dependent on program behavior and memory design choices.

Hierarchy Optimization

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

$$C_{\text{total}} = \sum_{i=1}^{n} c_i \cdot s_i$$

This implies that the cost is distributed over n levels. Since cl > c2 > c3 > … cn, we have to choose s1 < s2 < s3 < … sn.

The optimal design of a memory hierarchy should result in a T_{eff} close to the t_1 of M_1 and a total cost close to the cost of M_n.

The optimization process can be formulated as a linear programming problem, given a ceiling C_0 on the total cost— that is, a problem to minimize

$$T_{eff} = \sum_{i=1}^{n} f_i \cdot t_i$$

subject to the following constraints:

$$s_i > 0,\ t_i > 0 \quad \text{for } i = 1, 2, \ldots, n$$

$$C_{\text{total}} = \sum_{i=1}^{n} c_i \cdot s_i < C_0$$

4.4 Virtual Memory Technology

- To facilitate the use of memory hierarchies, the memory addresses normally generated by modern processors executing application programs are not *physical addresses*, but are rather *virtual addresses* of data items and instructions.
- Physical addresses, of course, are used to reference the available locations in the real physical memory of a system.
- Virtual addresses must be mapped to physical addresses before they can be used.

Virtual to Physical Mapping

- The mapping from virtual to physical addresses can be formally defined as follows:

$$f_t v = \begin{cases} m, & \text{if } m \in M \text{ has been allocated to store} \\ & \text{the data identified by virtual address } v \\ \varnothing & \text{if data } v \text{ is missing in } M \end{cases}$$

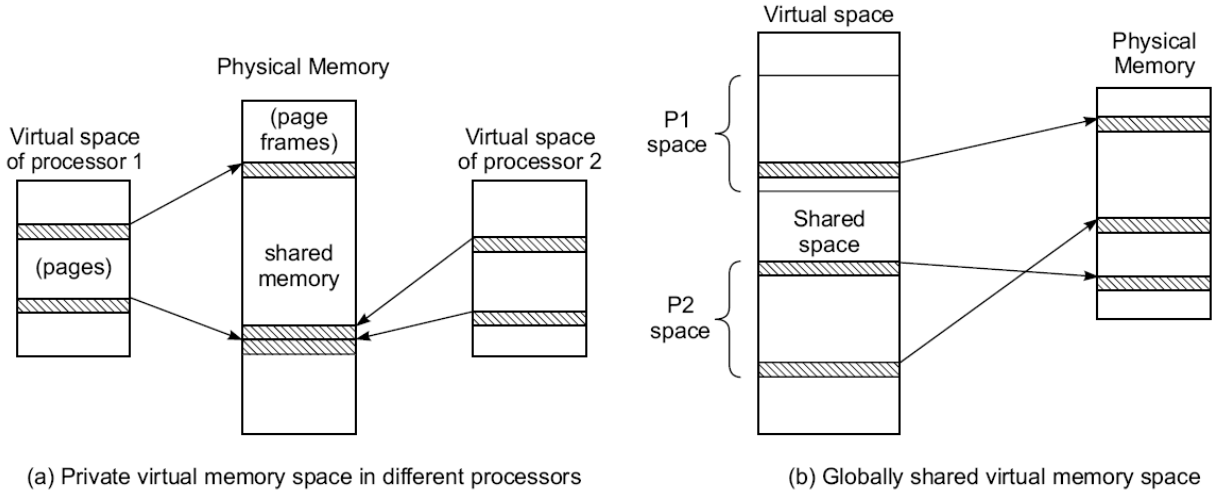
- The mapping returns a physical address if a *memory hit* occurs. If there is a *memory miss*, the referenced item has not yet been brought into primary memory.

Mapping Efficiency

- The efficiency with which the virtual to physical mapping can be accomplished significantly affects the performance of the system.
- Efficient implementations are more difficult in multiprocessor systems where additional problems such as coherence, protection, and consistency must be addressed.

Virtual Memory Models

1. Private Virtual Memory
2. Shared Virtual Memory



(a) Private virtual memory space in different processors

(b) Globally shared virtual memory space

Fig. 4.20 Two virtual memory models for multiprocessor systems (Courtesy of Dubois and Briggs, tutorial, *Annual Symposium on Computer Architecture*, 1990)

1. Private Virtual Memory

- In this scheme, each processor has a separate virtual address space, but all processors share the same physical address space.
- **Advantages**:
  - Small processor address space
  - Protection on a per-page or per-process basis
  - Private memory maps, which require no locking
- **Disadvantages**
  - The synonym problem – different virtual addresses in different/same virtual spaces point to the same physical page
  - The same virtual address in different virtual spaces may point to different pages in physical memory

2. Shared Virtual Memory

- All processors share a single shared virtual address space, with each processor being given a portion of it.
- Some of the virtual addresses can be shared by multiple processors.

Advantages:

- All addresses are unique

- Synonyms are not allowed

Disadvantages

- Processors must be capable of generating large virtual addresses (usually > 32 bits)
- Since the page table is shared, mutual exclusion must be used to guarantee atomic updates
- Segmentation must be used to confine each process to its own address space
- The address translation process is slower than with private (per processor) virtual memory

Memory Allocation

Both the virtual address space and the physical address space are divided into fixed-length pieces.

– In the virtual address space these pieces are called ***pages***.

– In the physical address space they are called ***page frames***.

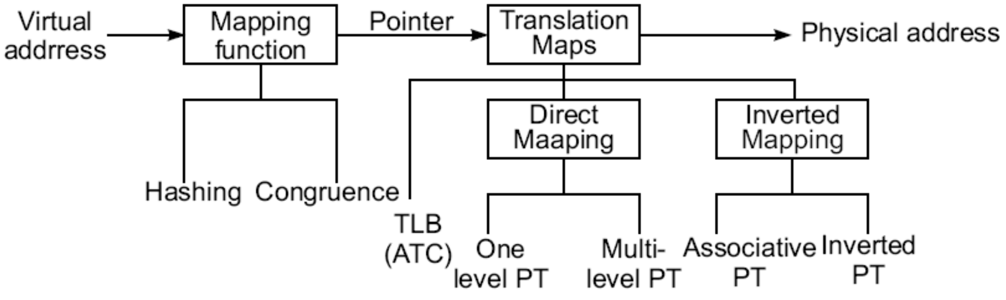
- The purpose of memory allocation is to allocate pages of virtual memory using the page frames of physical memory.

4.4.2 TLB, Paging, and Segmentation

Both the virtual memory and physical memory are partitioned into fixed-length pages. The purpose of memory allocation is to allocate pages of virtual memory to the page frames of the physical memory.

Address Translation Mechanisms

- The process demands the translation of virtual addresses into physical addresses. Various schemes for virtual address translation are summarized in Fig. 4.21a.



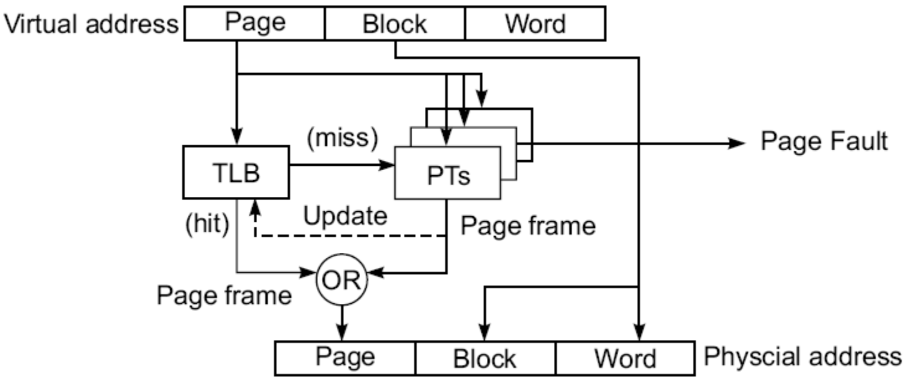
(a) Virtual addrress translation schemes (PT = page table)

- The translation demands the use of *translation maps* which can be implemented in various ways.
- Translation maps are stored in the cache, in associative memory, or in the main memory.

- To access these maps, a mapping function is applied to the virtual address. This function generates a pointer to the desired translation map.
- This mapping can be implemented with a *hashing* or *congruence* function.
- Hashing is a simple computer technique for converting a long page number into a short one with fewer bits.
- The hashing function should randomize the virtual page number and produce a unique hashed number to be used as the pointer.

Translation Lookaside Buffer

- The TLB is a high-speed lookup table which stores the most recently or likely referenced page entries.
- A *page entry* consists of essentially a (virtual page number, page frame number) pair. It is hoped that pages belonging to the same working set will be directly translated using the TLB entries.
- The use of a TLB and PTs for address translation is shown in Fig 4.21b. Each virtual address is divided into 3 fields:
  - The leftmost field holds the virtual page number,
  - the middle field identifies the cache block number,
  - the rightmost field is the word address within the block.



(b) Use of a TLB and PTs for address translation

- Our purpose is to produce the physical address consisting of the page frame number, the block number, and the word address.
- The first step of the translation is to use the virtual page number as a key to search through the TLB for a match.
- The TLB can be implemented with a special associative memory (content addressable memory) or use part of the cache memory.

- In case of a match (a hit) in the TLB, the page frame number is retrieved from the matched page entry. The cache block and word address are copied directly.
- In case the match cannot be found (a miss) in the TLB, a hashed pointer is used to identify one of the page tables where the desired page frame number can be retrieved.

Implementing Virtual Memory

There are 3 approaches to implement virtual memory:

1. Paging
2. Segmentation
3. A combination of the two called **Paged Segmentation**

1. Paging memory

- Memory is divided into fixed-size blocks called pages.
- Main memory contains some number of pages which is smaller than the number of pages in the virtual memory.
- **For example,** if the page size is 2K and the physical memory is 16M (8K pages) and the virtual memory is 4G (2 M pages) then there is a factor of 254 to 1 mapping.
- A page map table is used for implementing a mapping, with one entry per virtual page.

2. Segmented memory

- In a segmented memory management system the blocks to be replaced in main memory are potentially of unequal length and here the segments correspond to logical blocks of code or data.

For example, a subroutine or procedure.

- Segments, then, are ``atomic" in the sense that either the whole segment should be in main memory, or none of the segment should be there.
- The segments may be placed anywhere in main memory, but the instructions or data in one segment should be contiguous,

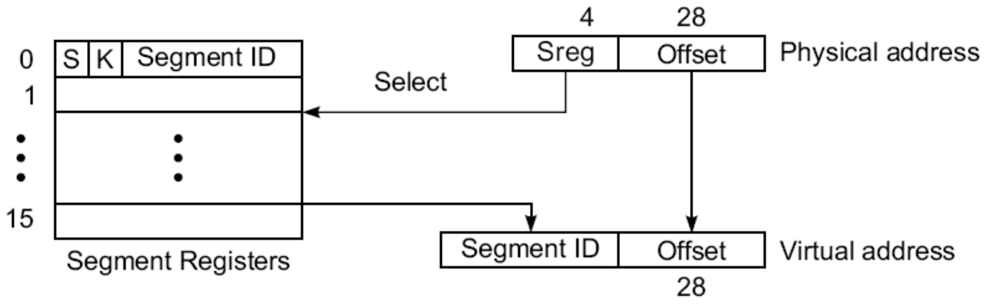
3. Paged Segmentation

- It is a combination of paging and segmentation concepts

- Within each segment, the addresses are divided into fixed size pages
- Each virtual address is divided into 3 fields
  - Segment Number
  - Page Number
  - Offset

Inverted paging

- Besides direct mapping, address translation maps can also be implemented with inverted mapping (Fig. 4.21c).
- An *inverted page table* is created for each page frame that has been allocated to users. Any virtual page number can be paired with a given physical page number.
- Inverted page tables are accessed either by an associative search or by the use of a hashing function.
- In using an inverted PT, only virtual pages that are currently resident in physical memory are included. This provides a significant reduction in the size of the page tables.
- The generation of a long virtual address from a short physical address is done with the help of segment registers, as demonstrated in Fig. 4.21c.



(c) Inverted address mapping

- The leading 4 bits (denoted *sreg*) of a 32-bit address name a segment register.
- The register provides a *segment id* that replaces the 4-bit sreg to form a long virtual address.
- This effectively creates a single long virtual address space with segment boundaries at multiples of 256 Mbytes (228 bytes).
- The IBM RT/PC had a 12-bit segment id (4096 segments) and a 40-bit virtual address space.
- Either associative page tables or inverted page tables can be used to implement inverted mapping.

- The inverted page table can also be assisted with the use of a TLB. An inverted PT avoids the use of a large page table or a sequence of page tables.
- Given a virtual address to be translated, the hardware searches the inverted PT for that address and, if it is found, uses the table index of the matching entry as the address of the desired page frame.
- A hashing table is used to search through the inverted PT.
- The size of an inverted PT is governed by the size of the physical space, while that of traditional PTs is determined by the size of the virtual space.
- Because of limited physical space, no multiple levels are needed for the inverted page table.

4.4.3 Page Replacement Policies

- Memory management policies include the allocation and deallocation of memory pages to active processes and the replacement of memory pages.
- Demand paging memory systems. refers to the process in which a resident page in main memory is replaced by a new page transferred from the disk.
- Since the number of available page frames is much smaller than the number of pages, the frames will eventually be fully occupied.
- In order to accommodate a new page, one of the resident pages must be replaced.
- The goal of a page replacement policy is to minimize the number of possible page faults so that the effective memory-access time can be reduced.
- The effectiveness of a replacement algorithm depends on the program behavior and memory traffic patterns encountered.
- A good policy should match the program locality property. The policy is also affected by page size and by the number of available frames.

Page Traces: A *page trace* is a sequence of *page frame numbers* (PFNs) generated during the execution of a given program.

The following page replacement policies are specified in a demand paging memory system for a page fault at time *t*.

(1) **Least recently used (LRU)**—This policy replaces the page in R(t) which has the longest backward distance:

$$q(t) = y, \quad \text{iff} \quad b_t(y) = \max_{x \in R(t)} \{b_t(x)\}$$

(2) **Optimal (OPT) algorithm**—This policy replaces the page in R(t) with the longest forward distance:

$$q(t) = y, \quad \text{iff} \quad f_t(y) = \max_{x \in R(t)} \{f_t(x)\}$$

(3) **First-in-first-out (FIFO)**—This policy replaces the page in R(t) which has been in memory for the longest time.

(4) **Least frequently used (LFU)**—This policy replaces the page in R(t) which has been least referenced in the past.

(5) **Circular FIFO**—This policy joins all the page frame entries into a circular FIFO queue using a pointer to indicate the front of the queue.

- An allocation bit is associated with each page frame. This bit is set upon initial allocation of a page to the frame.
- When a page fault occurs, the queue is circularly scanned from the pointer position.
- The pointer skips the allocated page frames and replaces the very first unallocated page frame.
- When all frames are allocated, the front of the queue is replaced, as in the FIFO policy.

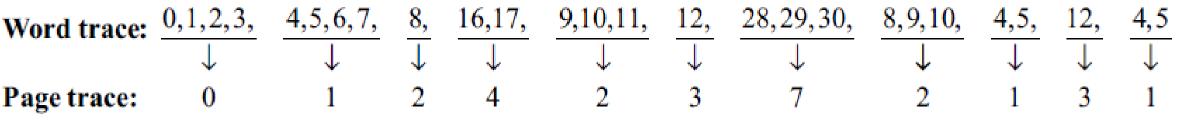
(6) **Random replacement**—This is a trivial algorithm which chooses any page for replacement randomly.

Example:

Consider a paged virtual memory system with a two-level hierarchy: main memory M_1 and disk memory M_2.

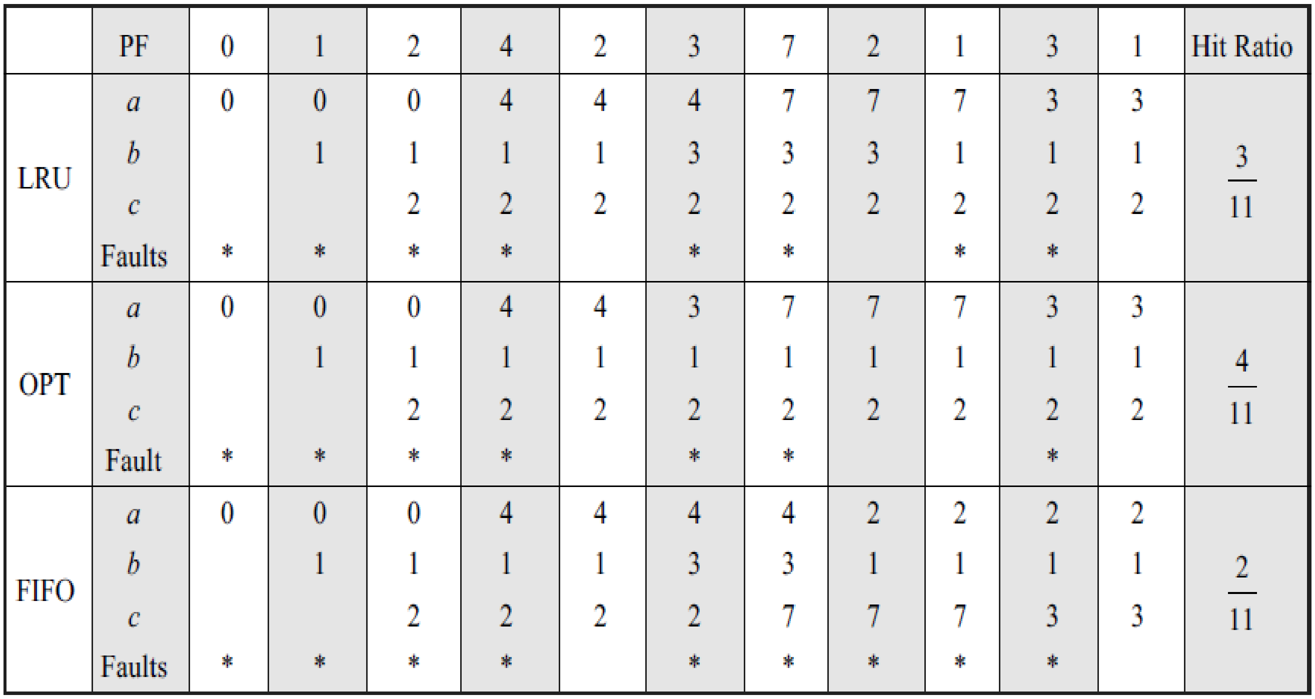
Assume a page size of four words. The number of page frames in M_1 is 3, labeled a, b and c; and the number of pages in M_2 is 10, identified by 0, 1, 2,….9. The ith page in M_2consists of word addresses 4i to 4i + 3 for all i = 0, 1, 2, …, 9.

A certain program generates the following sequence of word addresses which are grouped (underlined) together if they belong to the same page. The sequence of page numbers so formed is the *page trace*:



Word trace:	0,1,2,3,	4,5,6,7,	8,	16,17,	9,10,11,	12,	28,29,30,	8,9,10,	4,5,	12,	4,5
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
Page trace:	0	1	2	4	2	3	7	2	1	3	1

Page tracing experiments are described below for three page replacement policies: LRU, OPT, and FIFO, respectively. The successive pages loaded in the page frames (PFs) form the trace entries. Initially, all PFs are empty.



	PF	0	1	2	4	2	3	7	2	1	3	1	Hit Ratio
LRU	*a*	0	0	0	4	4	4	7	7	7	3	3	$\frac{3}{11}$
	b		1	1	1	1	3	3	3	1	1	1	
	c			2	2	2	2	2	2	2	2	2	
	Faults	*	*	*	*		*	*		*	*		
OPT	*a*	0	0	0	4	4	3	7	7	7	3	3	$\frac{4}{11}$
	b		1	1	1	1	1	1	1	1	1	1	
	c			2	2	2	2	2	2	2	2	2	
	Fault	*	*	*	*		*	*			*		
FIFO	*a*	0	0	0	4	4	4	4	2	2	2	2	$\frac{2}{11}$
	b		1	1	1	1	3	3	1	1	1	1	
	c			2	2	2	2	7	7	7	3	3	
	Faults	*	*	*	*		*	*	*	*	*		