Chapter One

Part1

Chapter one:
CPU Organization

The part of Computer that perform bulk of data processing operation is called the central processing unit / and is referred to as CPU. The CPU contains the hardware component for processing instruction and data, and we have viewed it essentially as "black box" and have considered its interaction with I/O and memory.

1-1 Model CPU Architecture

The major structural components of CPU are:

* Arithmetic logic unit (ALU)
* Control unit (CU)
* Registers

As show in Fig. 1

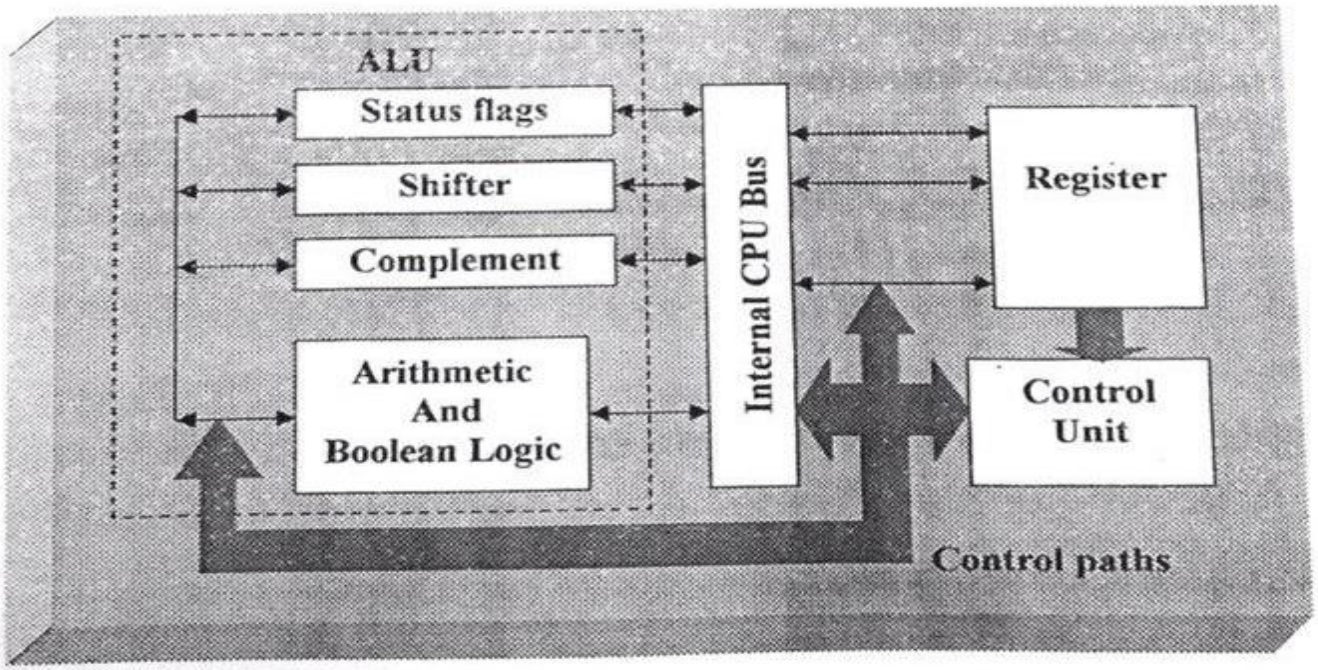


Fig. 1 Internal structure of CPU

***ALU:** The Arithmetic Logic Unit is the part of CPU which performs arithmetic and logic operation on data. All the other elements are mainly to bring data to ALU to process and take the result back out.

***CU:** The purpose of the control unit is to bring the instructions in from the main memory and then control their execution.

***Registers:** Small internal memory. CPU used registers because it needs to store instruction and data temporarily while an instruction is being execute.

There are some kinds of register as follows:

1-Accumulator: it is used as temporarily buffer to store input to ALU.

2-Data register: may be used only to holed data.

3-Condution code: they are a bits set by the CPU hardware as result of operation.

4-Program counter: contains of the address of an instruction to be fetch.

5-Instruction Register: contains the instruction must recently fetch.

6-Memory Address Register: contain the address of a location in memory.

7-Memory Buffer Register: contains a word of data to be written to memory or the word mast recently read.

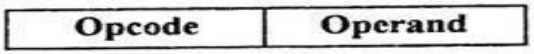
1-2 Instruction set design issues:

The operation of the CPU is determined by the instructions it executed. These are referred to as machine instructions or computer instructions. The CPU may perform a variety functions, and these are reflected in the variety defined for the CPU. The collection of different instructions that the CPU can execute is referred to as the CPU instruction set.

Each instruction must contain the information required by the CPU for execution (where the steps that involved in instructions are: fetch data, process data, store data). These elements are:

1- Operation code field: specifies the operation to be performed, and also known as opcode

2- Address field: designates a memory address or a processor register.



Computers may have instructions of several different lengths containing varying number of addresses. The number of address in the instruction format of a computer depends upon the internal organization of its registers. Most computers fall in one of three types of CPU org. :-

1- Single accumulator organization.

2- General register organization.

3- Stack organization.

For the first type of organization see the Fig. (4). Where all operations are performed with an implied accumulator register. The instruction format in this type of computer use one-address field for example:

ADD x

Which mean Acc ← Acc + [x] where Acc the accumulator and [x] the memory word located at the address x.

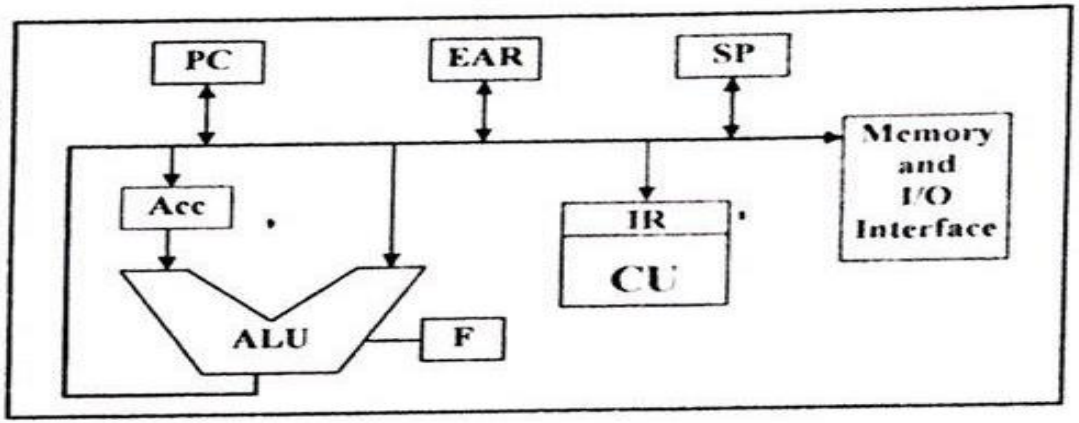


Fig. (4) Organization of an Accumulator based processor

An example of a general register type of organization is presented in Fig.5. The instruction format in this type of computer needs three register address fields, example:

ADD R1, R2, R3

Denote the operation $R1 \leftarrow R2 + R3$

Also this type of computer can support two address instructions if the destination reg. is the same as one of the source registers:

ADD R1, R2 denote R1←R1+R2

The reason for called this type of Org. general registers because any one of these reg. can be used to hold data, memory addresses, or the result of arithmetic or logic operation. Finally there are eight general register (R0 through R7)

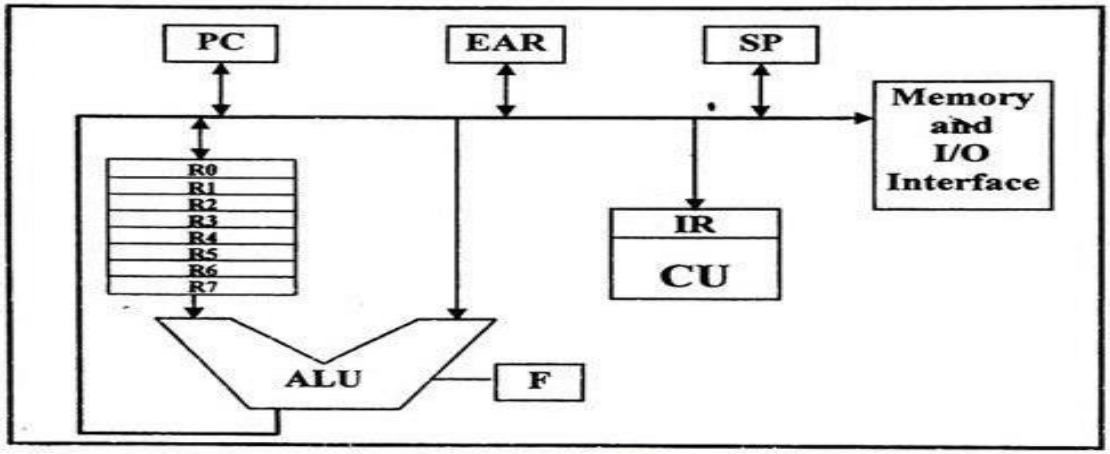


Fig. (5) Organization of general register type.

The stack organized CPU is presented in Fig (6). Computers with stack organization would have PUSH and POP instructions which require an address field. Thus, the inst. :-

PUSH x

Will push the word at address x to the top of the stack. The stack pointer is automatically updated. The operation type instructions

do not need address field in stack-organized computers. This is because the operation is performed on the two items which are on top of the stack. The inst. ADD here operation code only with no address field. This operation has the effect of popping the two top numbers from the stack, adding the numbers and pushing the sum into the stack. Then the zero-address-type instructions which are characteristic of a stack –organized CPU.

Note: - the register temp in Fig. (6) hold the first operand will be removed from the stack and the second operand will be directly routed from the stack to the right input of ALU.

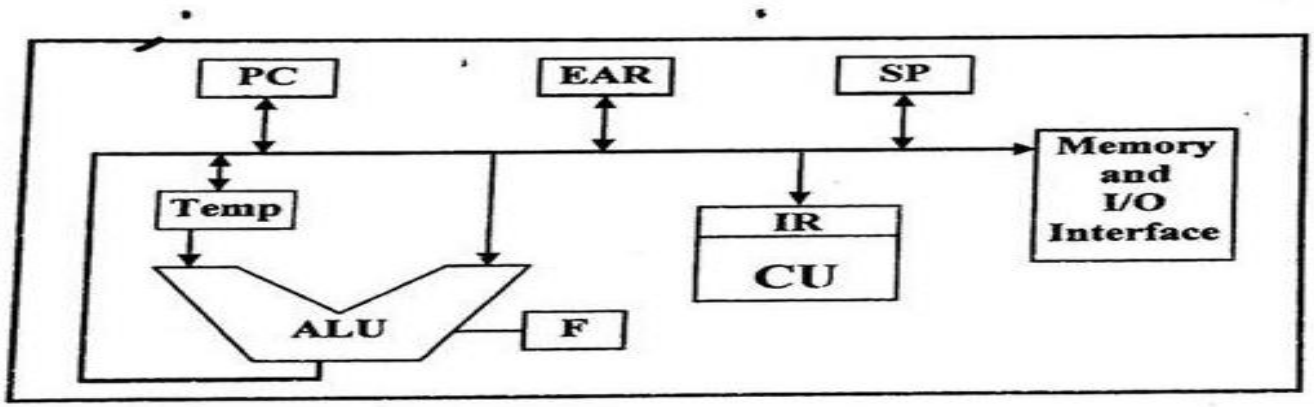


Fig. (6) Organization of the stack machine.

To illustrate the influence of the number of addresses on computer programs, we will evaluate the arithmetic statement:

$$X = (A + B) * (C + D)$$

Using zero, one, two, or three address instruction

Thank you …

Q&A

Chapter One

Part2

Three - address instructions:

ADD	R1, A, B	R1← [A]+[B]
ADD	R2, C, D	R2← [C]+[D]
MUL	X , R1 ,R2	[X]←R1* R2

Two - address instructions:

MOV	R1, A	R1← [A]
ADD	R1, B	R1← R1+[B]
MOV	R2, C	R2← [C]
ADD	R2, D	R2← R2+[D]
MUL	R1, R2	R1← R1 *R2
MOV	X , R1	[X]← R1

One - address instructions:

Load	A	ACC← [A]
ADD	B	ACC← ACC+[B]
Store	T	[T] ← ACC
Load	C	ACC←[C]
ADD	D	ACC←ACC+[D]
MUL	T	ACC←ACC*[T]
Store	X	[X]← ACC

Zero- address instructions

PUSH	A	TOS← A
PUSH	B	TOS← B
ADD		TOS← (A + B)
PUSH	C	TOS← C
PUSH	D	TOS← D
ADD		TOS← (C + D)
MUL		TOS← (C+D)*(A+B)
POP	X	[X]← TOS

(Where TOS stands for top of stack)

Addressing modes:

The operation field of an instruction specifics the operation to be performed. This operation must be executed on some data stored in computer registers or memory words .The way the operands are chosen during program execution is dependent on the addressing mode of the inst. The addressing mode specifics a rule for interpreting or modifying the address field of the inst. before the operand is actually referenced .Now below the most common addressing modes:

1- Immediate addressing mode:

In this mode the operand is specified in the inst. itself other words, an immediate–mode inst. has an operand field rather than an address field. (Here the data as part of the instruction). The data for the instruction MOV AX , 10F0 is supplied operand immediately after the instruction Opcode.

2- Direct addressing mode:

In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the inst. The instruction MOV AX, XRAY .mean store the contents of memory location XRAY in register AX.

^

3- Indirect addressing mode :

In this mode the address field of the inst gives the address where the effective address is store in memory. A few addressing mode require that the address field of the inst be added to the content of a specific register in the CPU. The effective address in these modes is obtained from the following :

Effective address=Address part of instruction + Content of CPU reg.

Ex: MOV AX, @ THERE

MOV AX, [1023+CX]

4- Register addressing mode:

In this mode the operands are in register which reside within the CPU.

Ex: - MOV DX , CX

5- Register-indirect addressing mode:

In this mode the instruction specifics a reg. in the CPU whose contents give the address of the operand memory. In other words the selected register contains the address of the operand rather than the operand itself.

MOV AX, [SI]

6-Relative Addressing mode:

In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address (EA) MOV AL, [PC + array]

7- Indexed Addressing mode :

In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.

Ex: MOV AL, [SI + array]

That mean EA = [SI] +array

8-Base-Register Addressing mode:

In this mode the content of base register is added to the address part of the instruction to obtain the EA. This similar to the indexed addressing mode except that the register is now called base register instead of an index register.

Ex: MOV [BX] + Beta, AL

Where EA = [BX] + Beta

Instruction types:

In general, the instruction set of different computers differ from each other mostly in the way the operands are determined from the address and mode fields. The actual operations available in the instructions set are not very different from one computer to another. Then the general types of instruction are found on all machine and may be broadly classified in to ~~six~~ seven groups, these are:

1- Data transfer instructions.
2- Arithmetic instructions.
3- Logic instructions.
4- Shift and rotate instructions.
5- Program control instructions.
6- System control instructions.
7- I /O instructions.

1- Data transfer instructions:

Data transfer instruction cause transfer of data from one place in the computer to another without changing the data content .The most common transfers are between:

1- Memory and registers.

2- Registers and registers

3- Memory and memory

4- Registers and input or output

The fallowing table (1) gives a list of eight data transfer instruction used in many computers.

Name	Function	mnemonic
Load	Transfer from memory to reg.	LD
Store	Transfer from reg. to memory	ST
Move	Transfer from (reg. & memory) or (reg. & reg.) or (memory & memory)	MOV
Exchange	Swap between (reg. & reg.) or (reg. & memory)	XCH
Input Output	⎫ Transfer among registers and Input or ⎭ Output terminal.	IN OUT
Push Pop	⎫ Transfer between registers and ⎭ memory stack.	PUSH POP

Table (1) typical data transfer instructions.

2- Arithmetic instructions:

The four basic arithmetic operations are adding, subtraction, multiplication, and division. Most computers provide a list of typical instructions is given in table (2).

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with barrow	SUBB
Negate (2's complement)	NEG

Table (2) typical arithmetic instructions.

Thank you …

Q&A

Chapter One

Part3

3- Logical instructions

Logical instructions perform binary operations on strings of bit. The logical instructions consider each bit of the operand separately and treat it as a Boolean variable. Some of typical logical operations is given in table (3)

Name	mnemonic	Example
And	AND	01101010.11110011=01100010
Or	OR	01101010+11110011=11111011
NOT	NOT	NOT(01101010)=10010101
Exclusive-or	XOR	01101010⊕11110011=10011001

Table (3) typical logical operations

4- Shift/Rotate instructions

The shift operations cause the bits of a word are shifted left or right, on one end, the bit shifted out is lost, on the other end a 0 is shifted in.

But the Rotate operations preserve all of the bits being operated on . Then here the bits shifted out at one end of the word are not lost as in a logical shift but are circulated back in to the other end .

The table (4) shows three types of these instructions:-

1- Arithmetic shift (left or right)

2- Logical shift (left or right)

3- Rotate shift (left or right) through or without the carry flag:-

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

Table (4) typical shift / rotate instructions

5- Program control instructions:

The program control instruction when execute may change the address value in the program counter and cause the flow of control to be altered. In other words, these instructions specify conditions for altering the content of the program counter. The change in the value of program counter as result of the execution of a program control inst; causes break in the sequence of inst. execution. These instructions may be classified into the following groups:

1- Unconditional branch instructions.

2- Conditional branch instructions.

3- Subroutines call instructions.

4- Interrupt-handling control instructions.

Some typical program control instructions are listed in table(5).

Name	Mnemonics
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Interrupt	INT

Table (5) typical program control instructions.

6- System control instructions:

System control instructions are generally privileged instruction that can be executed only while the processor is in certain privileged state or is executing a program in a special privileged area of memory. Typically, these instructions are reserved for the used of the operating system.

Some examples of system control operations:

1- A system control instructions may read or alter a control register.

2- Instruction to read or modify a storage protection key.

3- Access to process control blocks in a multiprogramming system.

The following some of these instructions:

Mnemonic	Name
WATT	Wait to synchronize
TAS	Test and set
LOCK	Lock of coprocessor
OSCALL	Causes interrupt

Table (6) some of system control instructions.

7- I/O instructions :

I/O instructions allow a processor to perform input and output operation. An input instruction allows a peripheral to transfer a word to either CPU register or memory. Similarly, an output instruction enables a processor to transfer a word to buffer register of a peripheral device. Then these instructions cause to exchange the data between the system and the peripheral, and table (7) show some example of this type of instructions.

Mnemonic	Name
START	Initialize I/O operation
TEST	Test I/O system
IN	Input from device
OUT	Output to the device

Table (7) some I/O instructions.

(Computer Architecture) ٤ – معمارية الحاسبات

CPU Organization :
A Model CPU Architecture , Instruction Set Design Issues , And Language Oriented Architectures .

Microprogramming :
Design Of CPU Control Unit , Microprogrammed Vs Hardwired Control , Complexity Of Microprograms , And Firmware .

I / O:
Peripheral Control Strategies .
Direct Memory Access , And I /O Channels .

Memory Management :
Register Windowing , Memory Interleaving , Cache Memory , And Tagged Storage .

Pipeline And Vector Processing :
Instruction Pipelining , Arithmetic Pipelining (Integer And Floating Point Multiplication), Systolic Arrays , And Vector Processing .

Multiprocessors :
Interprocessor Communication Networks And Methods And Cache Coherence .

Associative Memory :
Content – Addressable Memories , Arithmetic In Memory , applications (DataBase Machines) .
Non – Von – Neumann Architectures : Dataflow And Graph Reduction .

(References) المصادر

1- David A. Patterson And John L. Hennessy , " Computer Organization And Design " The Hardware / Software Interface . Morgan Kaufmann 1998 .

2- M.M Maro . " Computer Systems Architecture " , 3rd Ed. , Prentice – Hall , 1993 .

Thank you …

Q&A

Chapter Two

Part1

Chapter two

Microprogramming

Microprogramming is an implemented technique in which a simpler, cheaper, and faster computer is programmed at level below assembly language. If the computer is implemented in hardware with simple instruction dealing directly with data paths, registers, and memory and I/O device, we would use the term microprogramming.

2-1 Design of CPU control unit:

In Fig. (1) the general model of CU.

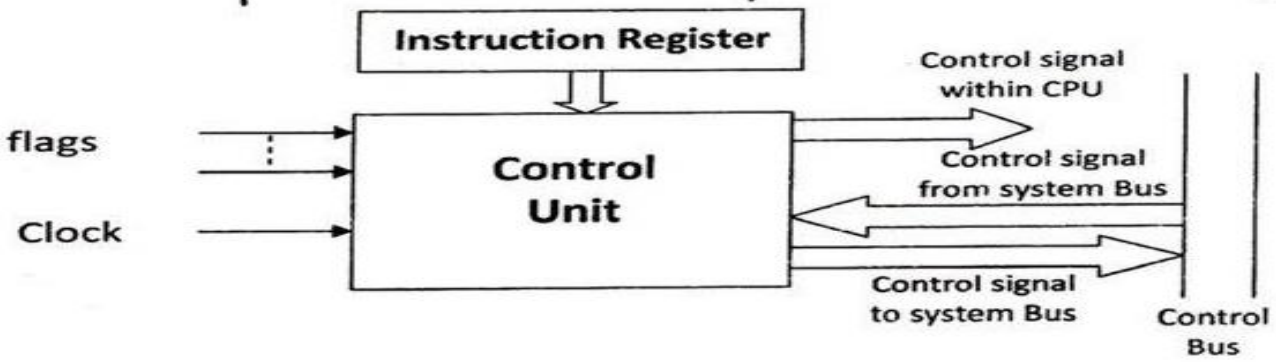


Fig. (1) model of the CU.

2-1 Micro-Operations:

Each operation of a computer in execute the program consist of a sequence of instruction cycles, with one machine instruction for each cycle. In fact, each of the smaller cycles involves of series of steps. These steps refer it as micro-operations. The prefix micro refers to the

fact that each step is very simple and accomplishes very little. Fig. (2) show the constituent element of program execution.

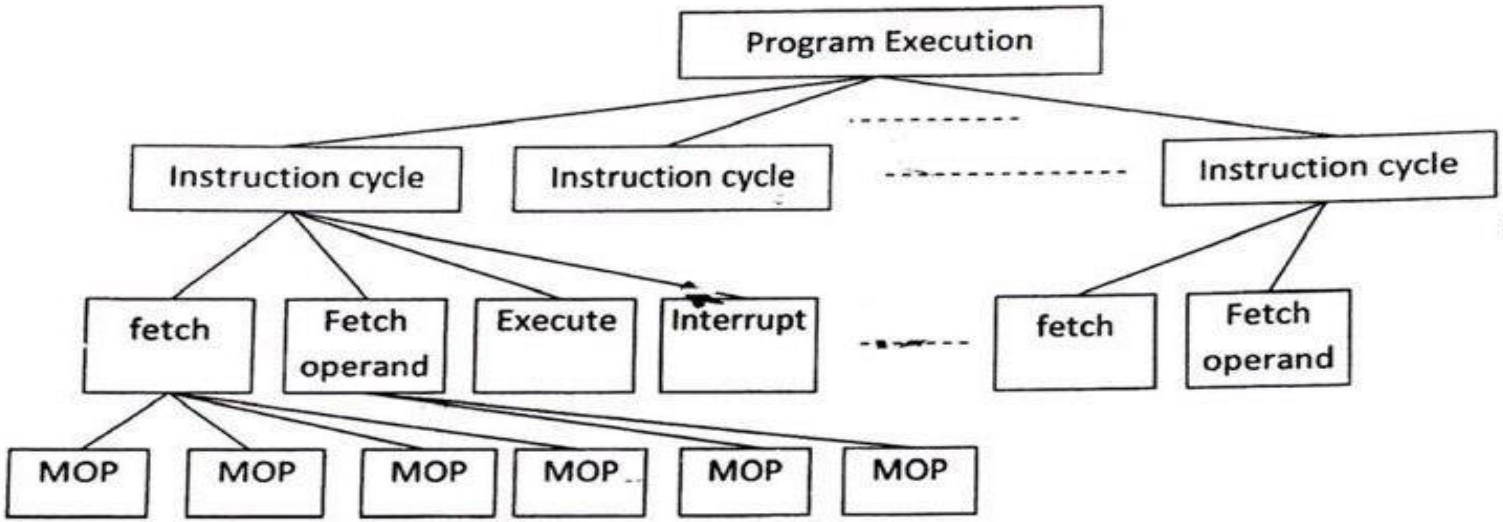


Fig. (2) constituent element of program execution.

{ MOP =Micro-Operation }

The fetch cycle: Occurs at the beginning of each instruction cycle and cause an instruction to be fetch from the memory. The simple fetch cycle consist of three steps and four micro-operations, several of them can take place during one step, and can write this sequence of events as follows (look Fig (3)):

T1: MAR ← [PC]

T2: MBR ← Memory

PC ← [PC] + 1

T3: IR ← [MBR]

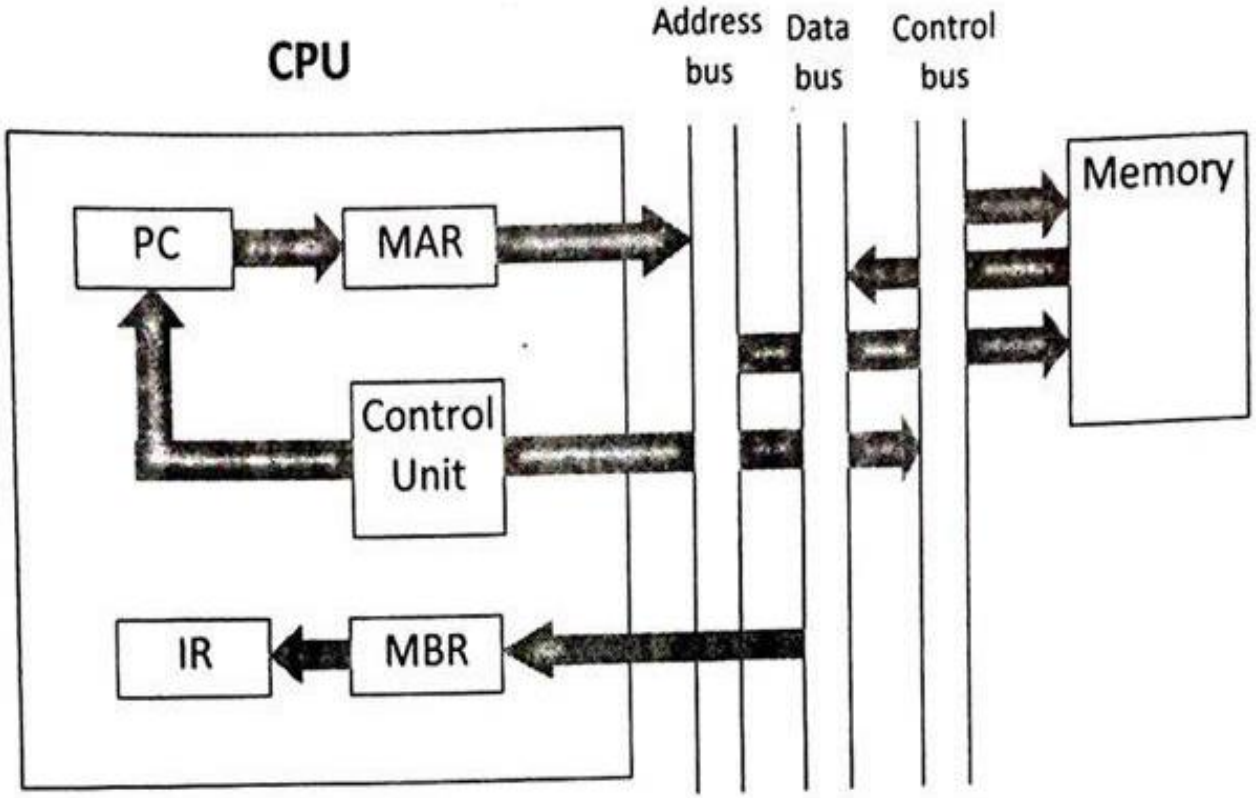


Fig. (3) Fetch cycle

The fetch operand cycle: the next step is to fetch source operand. And if assume one-address instruction format, the data flow that is indicate it in Fig. (4) include the following micro-operations (if the instruction specifies an indirect address):

T1: MAR ← [IR [address]]

T2: MBR ← memory

T3: [IR [address]] ← [MBR [address]]

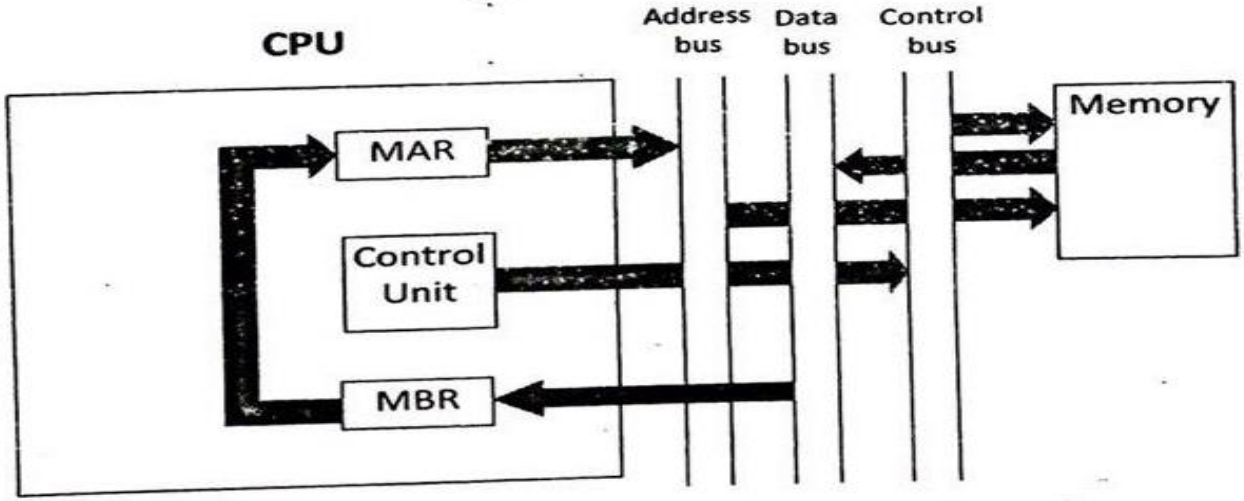


Fig. (4) Fetch operand

The Execute cycle: In this cycle there are different sequence micro-operations for different opcodes. For example the instruction:

ADD R1, x

The following sequence of micro-operation might occurs:

T1: MAR ← [IR [address]]
T2: MBR ← memory
T3: R1 ← [R1]+[MBR]

Also consider a subroutine call instruction. As an example:

BSA x (branch and save address instruction)

The following micro-operation need:

T1: MAR ← [IR [address]]
MBR ← [PC]
T2: PC ← [IR [address]]
memory ← [MBR]
T3: PC ← [PC]+1

The interrupt cycle: Atthe completion of the execute cycle, a test is made to determine whether any enabled interrupts have occurred. If so, the interrupt cycle occurs. The nature of this cycle varies greatly from one machine to another. We present a very simple sequence of events, as illustrated in Fig. (5):

T1: MBR ← [PC]

T2: MAR ← save-address

PC ← Routine-address

T3:Memory← [MBR]

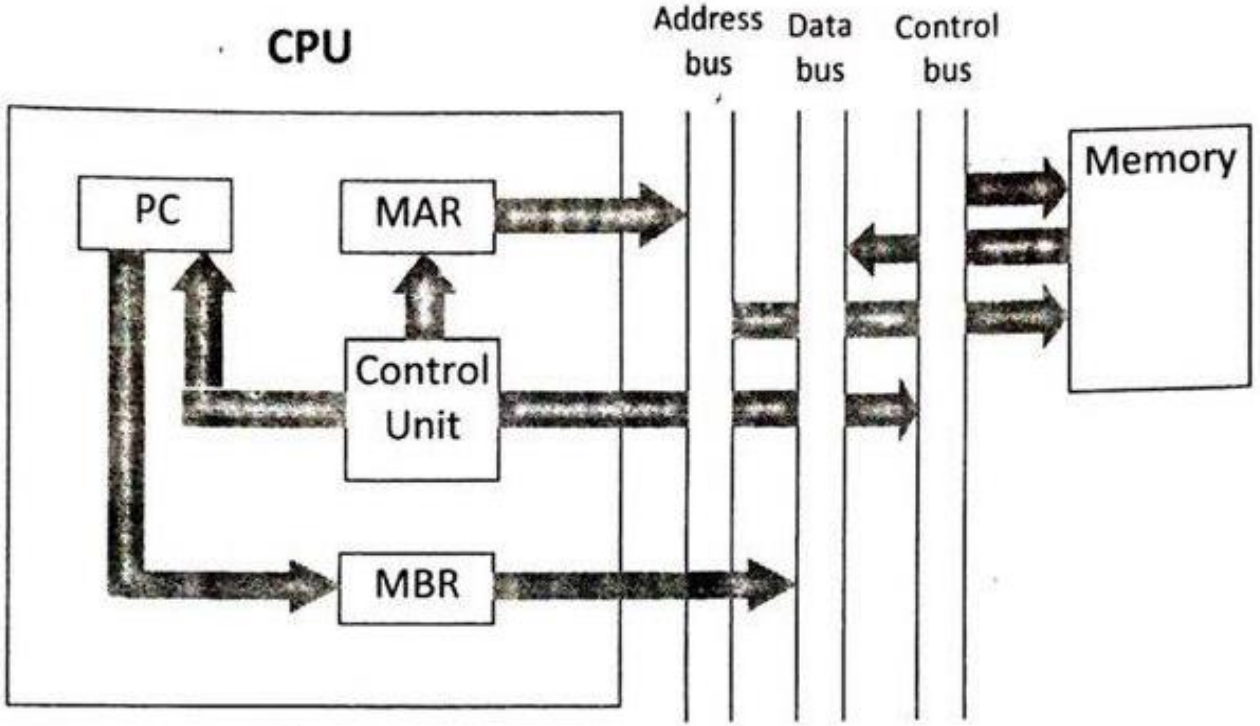


Fig. (5) Interrupt cycle

2-2 Control of the CPU: (functional Requirements)

There are some functions that the CPU must be perform which are called "functional Requirements". And a definition of these functional Requirements is the basis for the design and implementation of the CU. The following three steps process leads to characterization of the CU:

1- Define the basic elements of the CPU.

2- Describe the micro-operations that CPU perform.

3- Determine the functions that the CU must to be perform to cause the micro-operations to be performed.

We have already explain step1 and step2. For the step3 the CU perform two basic tasks:

1-Sequencing: cause the CPU to step through a series of micro-operations based on the program being executed.

2-Execution: cause each micro-operation to be performed internally, the CU must have the logic required to perform its sequencing and execution functions .

The CU must have inputs are:

{clock, IR, flags, control signals from control BUS}

And output are:

{control signals within the CPU ,control signal to control BUS}

2-3 Design methods:

Control units are designed in two different ways:

- Hardwired approach.
- Microprogramming

In the first approach , the control unit is essentially a combination circuit , its input logic signals are transformed in to a set of input logic signals , which are the control signals. In the microprogramming Approach , all control functions that can be simultaneously activated are grouped to form control words stored in a separate memory called the control memory. The control words are fetched from the control memory and the individual control fields are routed to various functional units to enable appropriate gates. When these gates are activated sequentially the desired task is performed.

Thank you …

Q&A

Chapter Two

Part2

2-3-1 Hardwired Implementation:

The control unit operates on the OPcode, and will perform different actions for different instructions. There should be a unique logic for each OPcode. This function can be performed by the decoder, in general the decoder will have n binary Input and 2^n binary output. Each of the 2^n different input patterns will activate a single unique output. The clock portion of the Cu issues a repetitive sequence of pulses. This is useful for measuring the duration of Micro-operations. However, the cu emits different control signals at different time units within a signal instruction cycle. Thus we would like a counter as input to the Cu, T1, T2..... At the end of an inst. cycle the Cu must feed back to the counter to be-initialize it at T1. Look for figure (6)

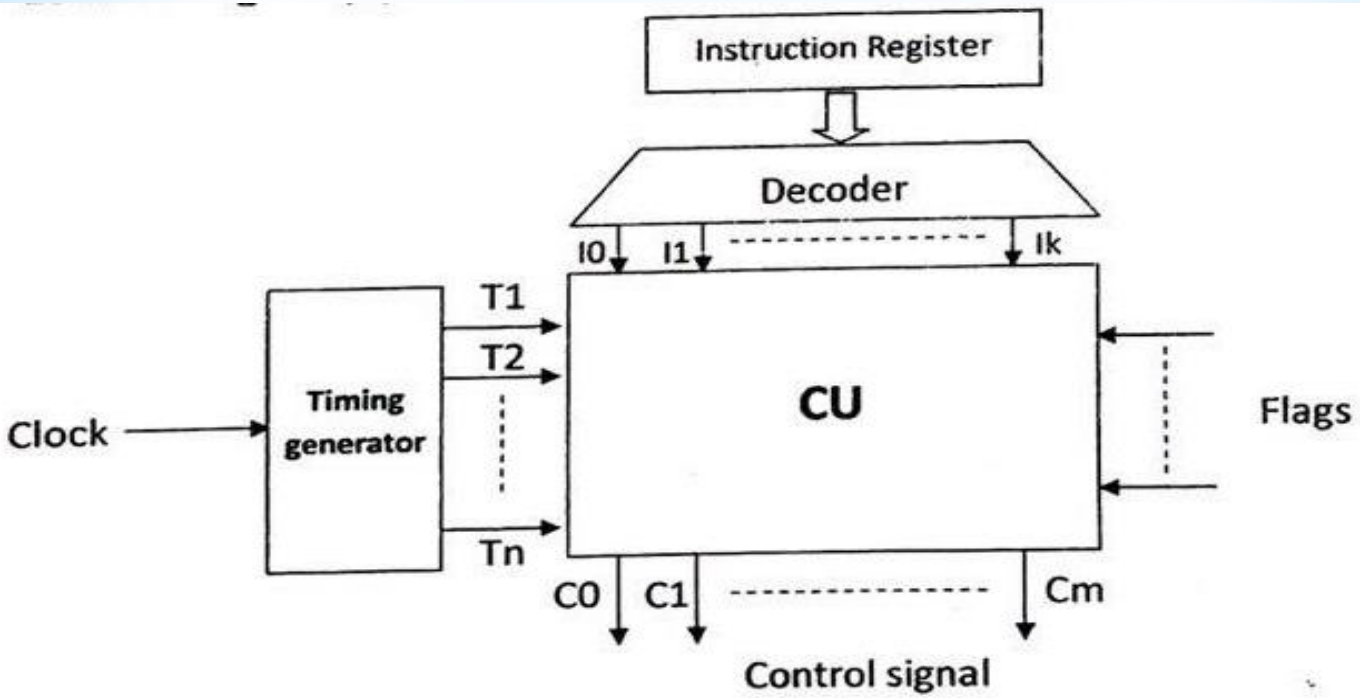


Fig. (6) Control unit with decoded inputs

The important thinks in the hardware approach the internal logic of the CU that produces output control signal as a function of its input signals. For each control signal there is a Boolean expression as a function of the inputs. These Boolean equations that define the behavior of the CU and hence of the CPU.

In modern complex CPU, the number of Boolean equations needed to define the CU is very large. The task of implementing the combination circuit that satisfies all of these equations become extremely difficult.

2-3-2 micro-programmed implementation:

Some time that is difficult to design and test piece of hardware. Also, the design is relatively inflexible (for example, it is difficult to change the design if one wishes to add a new machine instruction). Then, there is an alternative that is implementing a micro-programmed unit.

Consider that for each micro-operation the CU generate a set of control signal. Thus, for any micro-operation, each control line emanating from the CU is either on or off. This condition can represent by a binary digit for each control line. So, we could construct control word (CW) in which each bit represent one control line. Then, each micro-operation would be represented by different pattern of 1 and 0 in the control word.

Note: the term micro-instruction refers to the micro-operation occurring at one time and a sequence of these instructions is known as micro-programming or firmware.

The control word put in memory and has a unique address and also add address field to each CW, to indicating the location of the next CW to be executed. The result is known as a horizontal micro-instruction and Fig. (7) show the format of the microinstruction or CW.

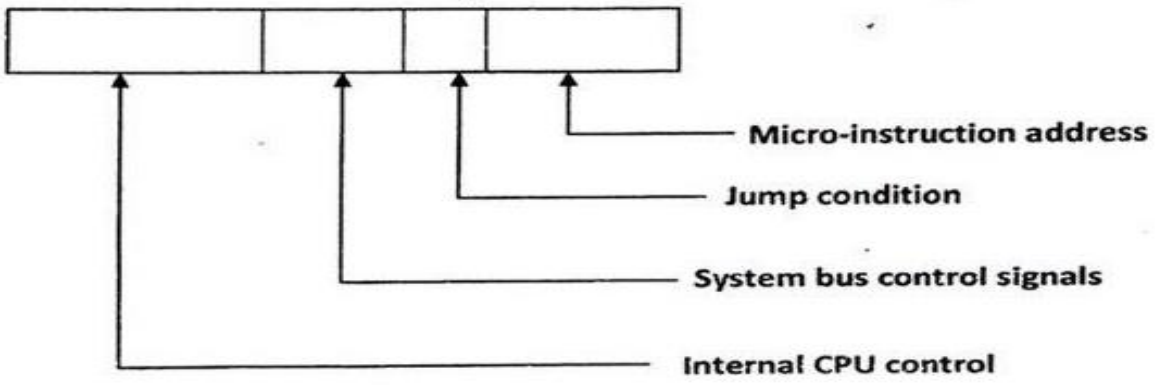


Fig. (7) Control word (horizontal micro-instruction)

There are one bit for each internal CPU control line and are bit for each system bus control line. There is a condition field indicating the condition and which the should be a branch, and there is a field for the address of the next micro-operation to be executed.

These CWs could be arranged in a control memory (CM) which contains a program that describes the behavior of the CU. That means we could implement the CU by simply executing that program. The Fig. (8) shows the key elements of such implementation. The set micro-instructions are stored in the control memory (CM). The control address register (CAR) contain the address of the next micro-instructions to be read, and the control buffer register (CBR) is to transferred to it the micro-instructions that read from CM. then the sequencing logic unit (SLU) to load the CAR and issues a read command. The following steps describe the control unit function:

1- the SLU issues a read command to the CM.
2- the word whose address in the CAR is read into the CBR.
3- The content of the CBR generate control signals and next address information for the SLU.
4- The SLU loads new address into the CAR based on the next

address information from the CBR and the ALU flags.

In Fig. (8) the upper decoder translate the opcode of the instruction into the CM address, where it is used for horizontal micro-instructions, but the lower decoder is used for vertical micro-instructions as appear in Fig. (9), where here the code is used for each action to be perform, and the lower decoder translates this code into individual control signals.

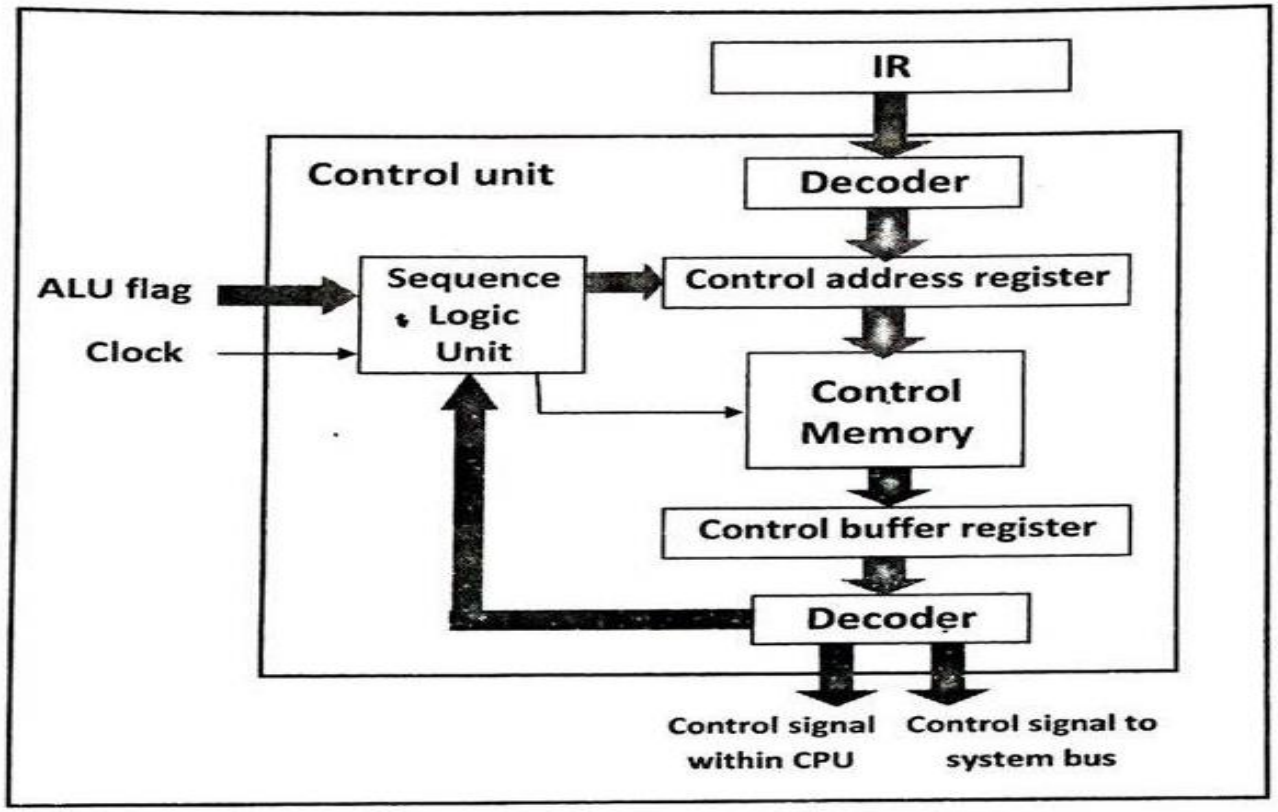


Fig. (8) Functioning of micro-programmed control unit

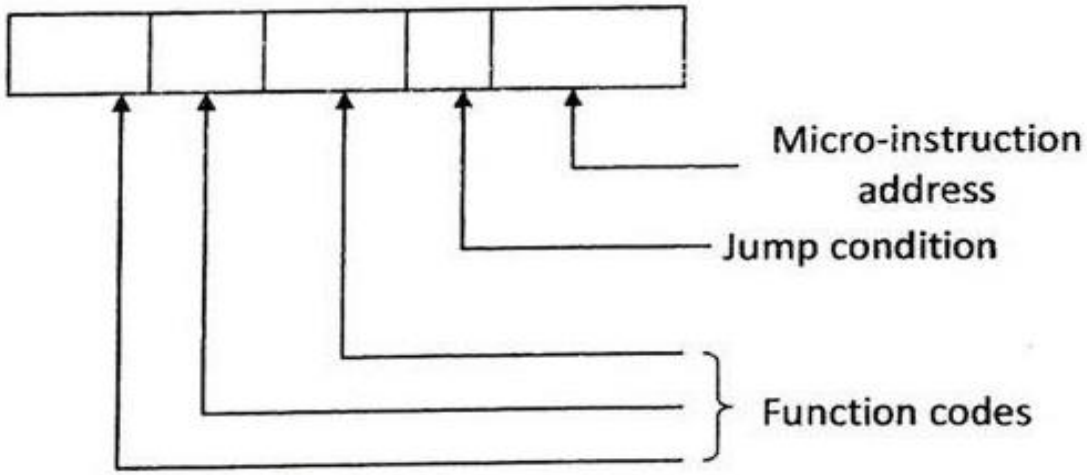


Fig. (9) Vertical micro-instruction.

2-3-3 Advantages and disadvantages:

The principle advantage of the used of micro-programming to implement a CU is that it simplifies the design of the CU. Thus it is both cheaper and less error to implement. A hardware CU must contain complex logic for sequencing through the many micro-operations of the instruction cycle. On the other hand, the decoder and sequencing logic unit of a micro-programming.

The principle disadvantage of micro-programmed unit is that it will be somewhat slower than a hardware unit. Despite this, micro-programming is the dominant technique for implementing CU in contemporary computers, due to its ease of implementation.

Thank you …

Q&A

Chapter Three

Part1

Chapter Three

Memory management

Memory is that part of computer system that used for the storage and then retrieval of data and instruction. The memory system cost is the significant fraction of the cost of the total system. The system performance is largely dependent on the organization storage capacity. And speed of operation of the memory system. Computer memory system can be logically divided into three groups:

1- **Internal memory (storage in CPU):** This refers to the set of registers in the CPU. Also the CU may require its own memory.

2- **Primary memory:** is the storage area in which all programs are executed. The CPU can directly access only these items that are stored in primary memory. Therefore all programs and data must be within the primary memory to speed up execution.

3- **Secondary memory:** refer to the storage medium comprised of slow device such as magnetic tapes and magnetic disks. These devices are used to hold data files and programs. Such as compilers and data base

management systems, which are not frequently needed by the processor. Called also (Auxiliary memory).

3-1 Characteristics of memory system:

The complex subject of computer memory is made more manageable if we classify memory systems according to their key characteristics:

- **Location:** as we say there are memory internal computer (in CPU and main memory) and external (secondary memory).
- **Capacity:** internal memory is expressed in terms of bytes or words (common word lengths are 8, 16, and 32 bits). External memory is expressed in bytes.

- **Unit of transfer:** the unit of transfer is equal to the number of data lines into and out of the memory module, this is often equal to the word length (this is the number of bits read out or written into memory at a time.
- **Access method:** there are four types which is:

  1- sequential access,

  2- Direct access,

  3- Random access,

  4- Associative access.

➢ **Performance**: Three performance parameters are used :

1-Access time: this is the time is taken to perform read or write operation (Random access). Or the time is take position the read-write mechanism at the desired location.

2- memory cycle time: Access time + any additional time required before the second access can commence.

3- Transfer rate: the rate at which data can be transferred into or out of a memory unit. For random-access memory it is equal to 1/(cycle time). But for non-random access memory is:

$$T_N = T_A + N/R$$

Where:

T_A = Average time to read or write N bit.

N = Number of bits.

R = Transfer rat in bit per second.

3-2 The memory Hierarchy

Most computers would run more efficiently if they were equipped of with additional storage to the capacity of the main memory. However, to meet the performance requirements, the designer needs to used expensive and lower capacity memory with goals can not rely one a single memory, but employ memory hierarchy as shown in Fig.(1).

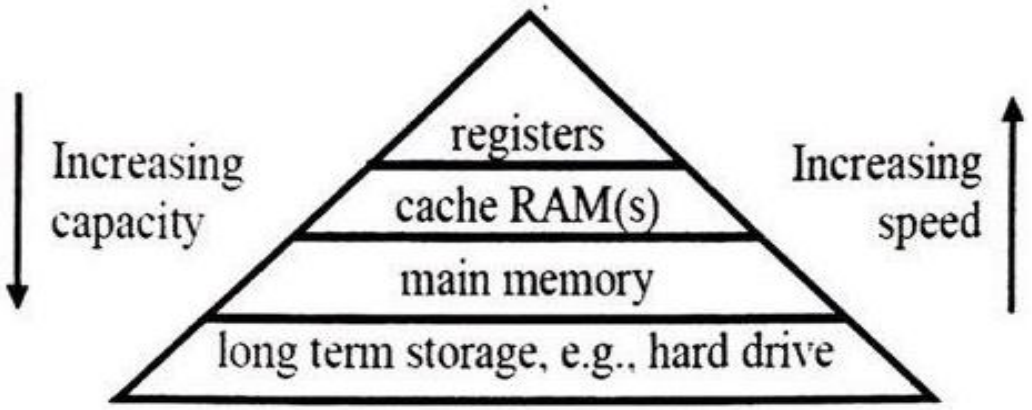


Figure 1 Block Diagram of a Standard Memory Hierarchy

If we move one this technology from top to down the following occur:

(a) Increasing in cost, (b) Increasing capacity,

(c) Increasing access time, (d) Decreasing frequency of access of the memory by the CPU.

The hierarchical memory is organized in four level as shown in Fig.(1), where the CPU directly Communicates with level 1 and level 1 communicate with level 2 and so on. In general all information accessed by the CPU is found in level 1, and a copy of all information will be held in the lowest level, if the required word is not find in level 1 then it is searched for it in the lower levels. Suppose the missing word is found in level j , where $j > 1$, it is then transferred to level j-1 and from level j-1 to level j-2 and so on, until it reaches to level 1. This transfer is essential because the CPU can access only level 1.

3-3 memory interleaving:

To overcome the different in speed between the memory and the CPU, employed many modules of memory that allows overlap or simultaneous access of memory cells in different modules. The characteristic of memory modules in a manner that allows access overlap is called interleaving. In the interleaving technique, the words in the modules are arranged so that N sequential ~~modules are arranged so that N sequential~~ بازی address a, a+1, a+2, ..., a+N-1 fall in N distinct modules, then the method for distribute the sequential address for sequential words on the different modules is following :

M = is the number of modules

L = number of addresses in the module (Max)

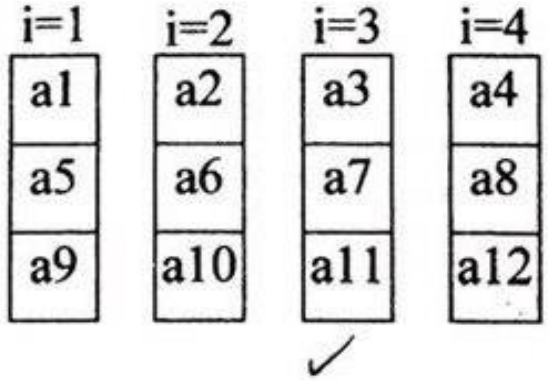
i = number of the current modules

Then can be arrange the sequential address over module as:

$K * M + i$

Where $0 \leq K \leq L-1$, $1 \leq i \leq M$

Example: M = 4 , L = 3 , i= 3 find the sequence addresses in this module:



K=0 → 0*4+3 → a3

K=1 → 1*4+3 → a7

K=2 → 2*4+3 → a11

3.4 Cache memory:

This memory is a fast and small memory that resides between main memory & the CPU as illustrated in Fig.(2). Where the active portions of the programs and data are placed in it, then the average memory access time can be reduce.

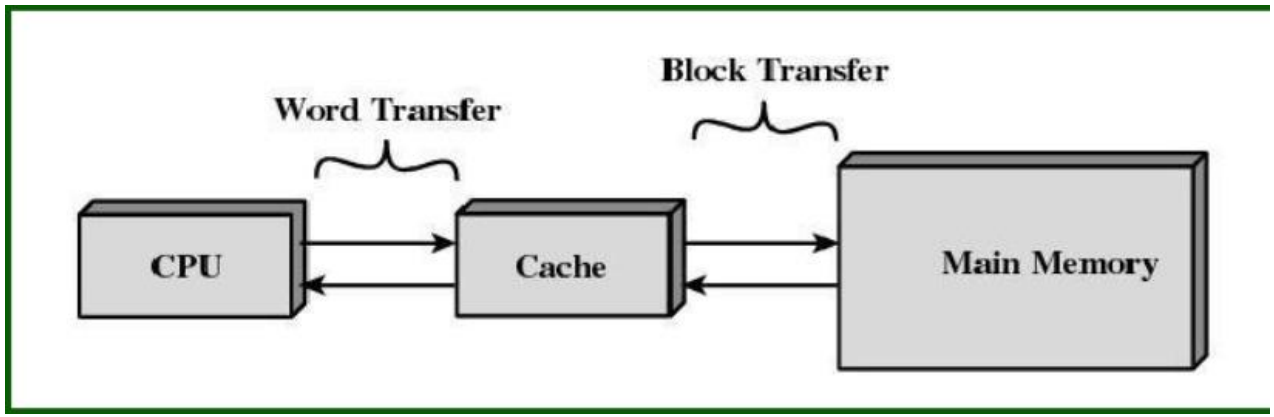


Figure 2 : Cache and Main memory

The cache memory access time is less than the access time of main memory by factor of 5 to 10. Then the fundamental idea of cache organization is that by keeping the most frequently accessed instructions and data in the fast cache memory. The average memory access time will approach the access time of the cache. Although the cache is only a small fraction of the size of main memory, a large fraction of memory requests will be found in the cache memory because of the locality of reference property of programs. Fig.(3) show the structure of cache and main memory system. Memory consists of up to 2^n addressable word. For mapping ~~purposes~~ process, this memory is considered to consist of a number of fixed length blocks of K word in each that is , there are $M = 2^n / K$ blocks.

Cache consist of C slots of K words in each and (C <= M). At any time, some subset of the blocks of memory resides in slots in the cache. If a word in a block of memory is read, that block is transferred to one of the slots of the cache. Since there are more blocks than slots. Each slot includes a tag that identifies which particular block is currently being stored. The tag is usually a portion of the main memory address.

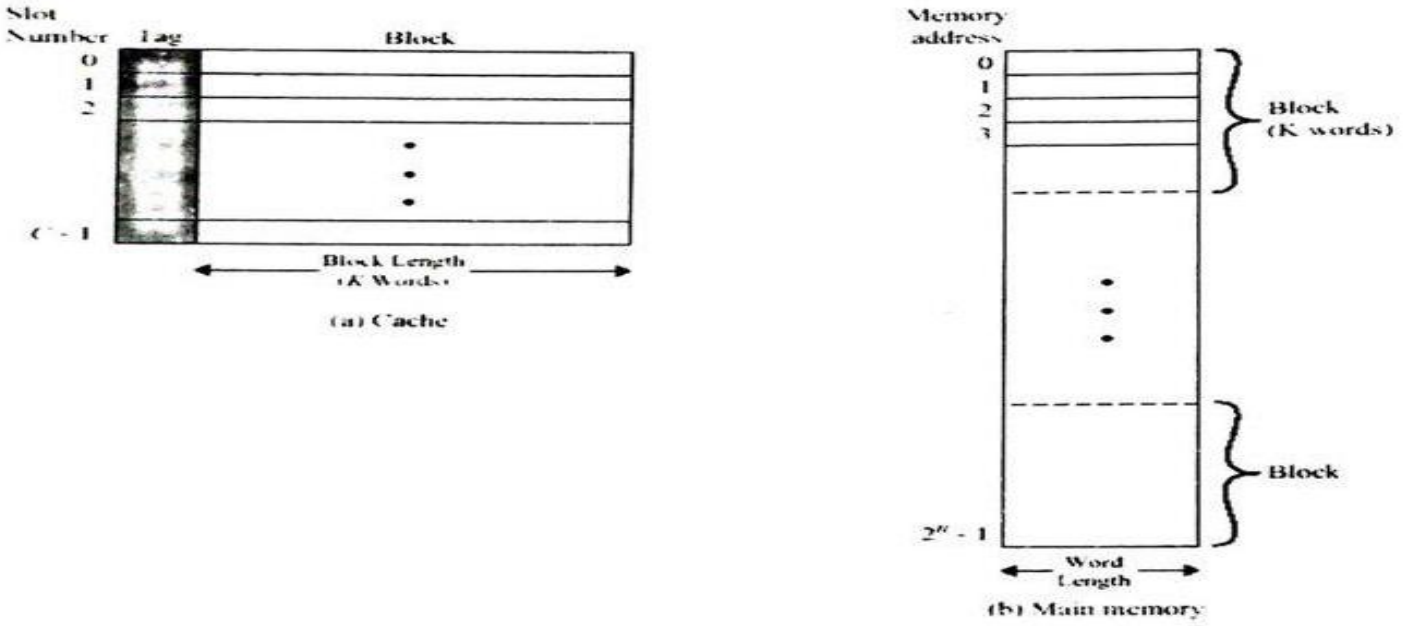


Figure 3: Cache/Main Memory Structure

3.4.1 Cache operation – overview

- CPU requests contents of memory location
- Check cache for this data
- If present, get from cache (fast)
- If not present, read required block from main memory to cache
- Then deliver from cache to CPU
- Cache includes tags to identify which block of main memory is in each cache slot

Thank you …

Q&A

Chapter Three

Part2

3.5

Mapping process:

The transformation of data from main memory to cache memory is referred to as mapping process. There are three types of mapping procedures are interest:

1. Associative mapping:

The associative memory stores both the address and content (data) of the memory word, as in figure (4). The diagram shows three words stored in the cache. The address value of 15 bits is shown as five-digit octal number and its corresponding 12 bit word is shown as 4 digit octal number. The CPU address of 15 bits is placed in the argument register and the associative memory is searched for matching address. If the address is found than the corresponding 12 bit data is sent to CPU. Otherwise, if no match, the main memory is accessed for the word.

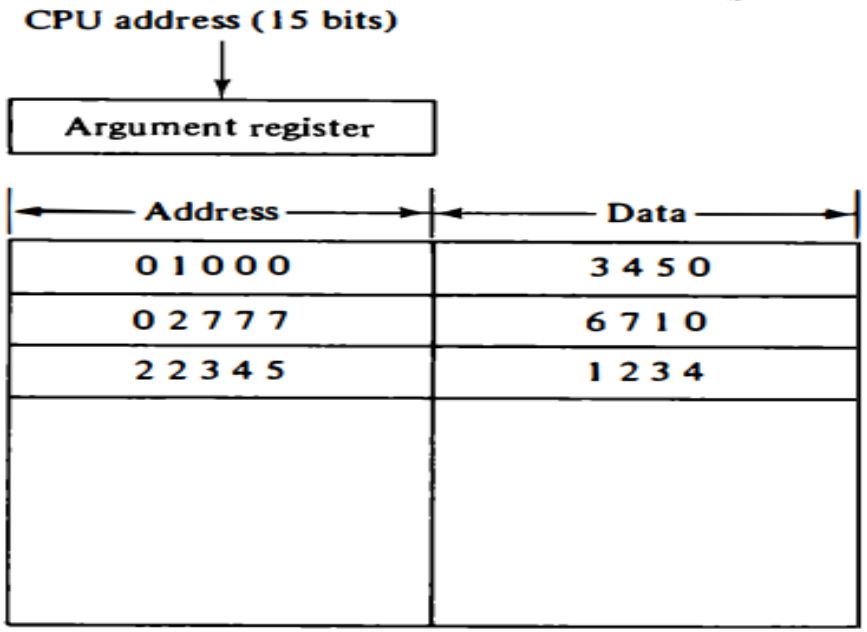


Fig. (4) Associative mapping cache (all numbers in octal)

2. **Direct Mapping:** The possibility of using random-access memory is shown in figure (5). The CPU address of 15 bits is divided into two fields. The 9 least significant bits is the index field and the remaining six bits form the tag field. Then cache word in cache consists of the data and its associated tag. Where the main memory needs address include both the tag and the index bits and the No. of bit of Index field is equal to the No. of address bits to access the cache when the CPU generates memory request, the Index field is used to access the cache and the tag field compare with tag in the word read from the cache, if match there is hit and the desired word data in the cache, if no match (miss) this required word is read from main memory and then stored in the cache together with the new tag.

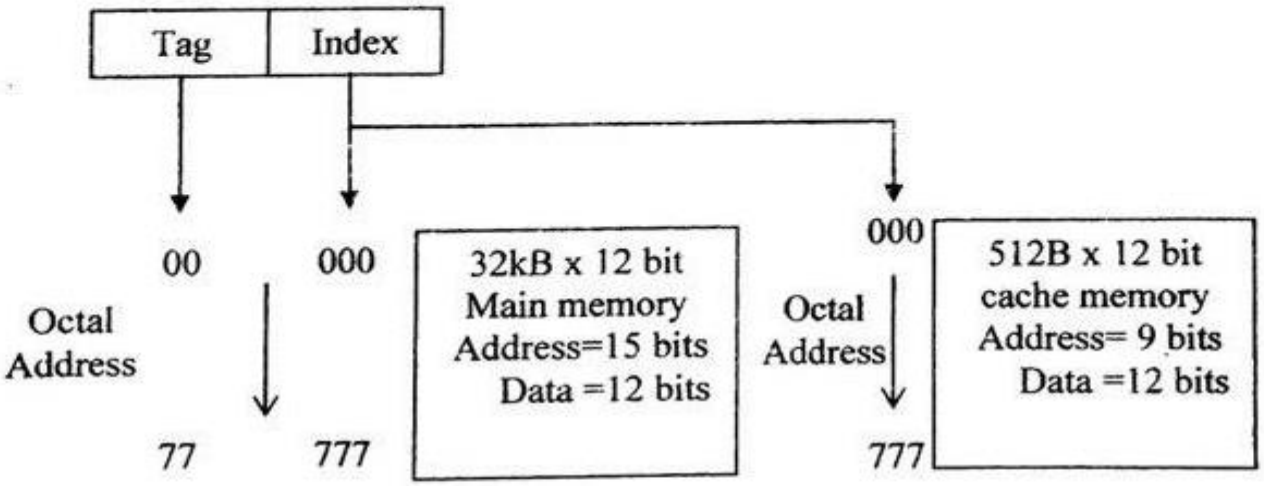


Fig. (5) Addressing relation ship between main and cache

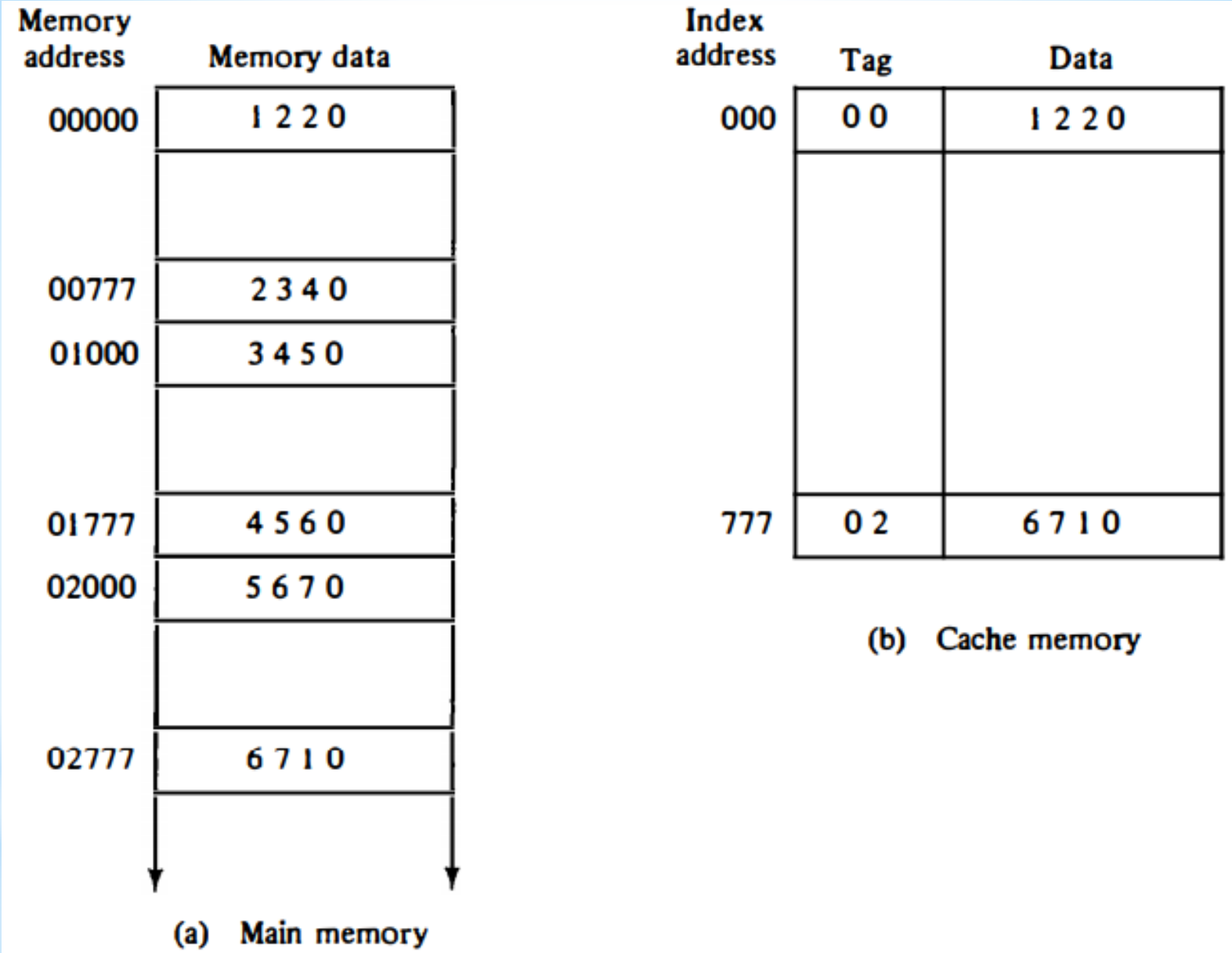


Figure (5) Direct mapping cache organization)

3. **Set-Associative mapping:** Here each data word is stored together with its tag and the No. of tag-data items in one word of cache is said to form a set. An example of a set-Associative cache organization for a set size of two is shown in figure (6). Each Index address refers to two data words and their associated tags. Each tag requires six bits and each data word has 12 bits, so the word length is $2x(6+12) = 36$ bits. An Index address of nine bits can accommodate 512 words. Thus the size of cache memory is 512×36. See that in fig (6) the words stored at addresses 01000 and 02000 of main memory are stored in cache at index address 000, also similarly the words at address 02777 and 00777 are stored in cache at Index address 777. When the CPU make memory request, the Index used to access the cache and the tags field of the CPU address is then compare with both tags in the cache to determine if a match occur.

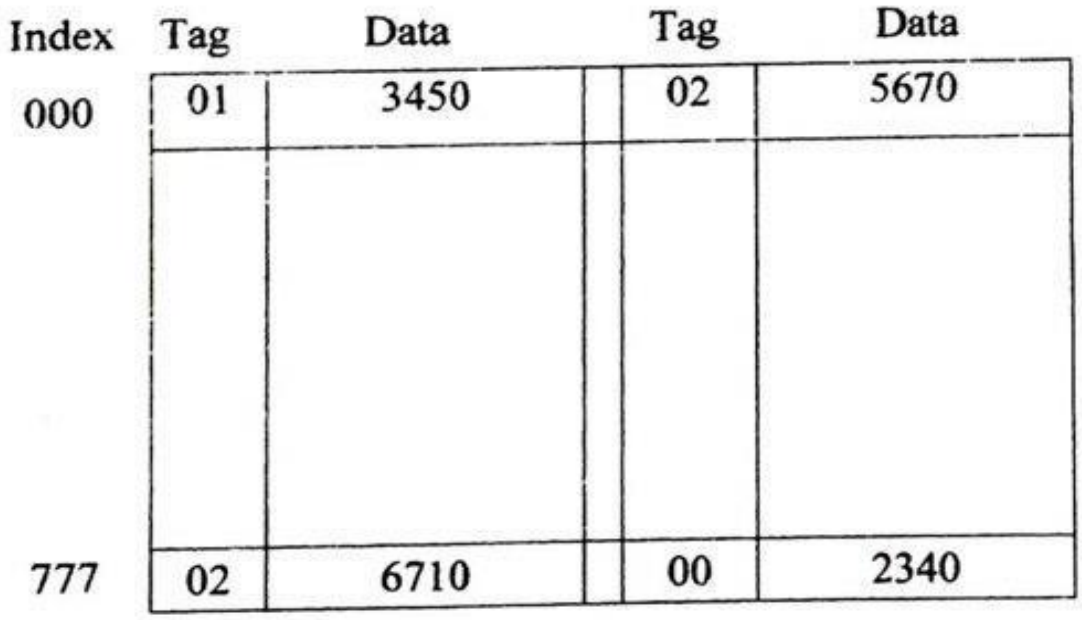


Fig. (6) Two way set associative mapping cache

4. **Tagged Storage:** Often each memory cell has bits associated with it. For special meanings. This bits is propagated and checked by hardware circulars as a means of verifying the correct retention of information in the memory cell.

In practice, tagging of each memory location in main memory. For conveying information as (1) Instruction /data. (2) Defined /undefined, (3) Read-Only/ read-write, (4) primitive data type.

The tagged store increases the width of each memory cell and, thus, increases total memory size. If tags exist, it is implied that the logic circuits to check data will exist.

If tagged storage is implemented to include not only main memory but follows the data into cache memory, stacks, registers, etc. then the errors arising from undefined data, wrong type would be eliminated.

Microcomputer Memory:

There are two types of memory chips used in microcomputer system. RAM (random Access memory) and Rom (Read only memory). RAM is used for storing data, variable parameters, and intermediate results that need updating and are subject to change. Rom is used for storing programs and table constants that do not change in value one the production of the microcomputer system is complete.

Ram and RoM Chips:

The block diagram of RAM chip is shown in fig (7). The capacity of memory is 128 words of 8 bits in each. This required 7 bit address and 8 bit bidirectional data bus. The read and write inputs specify the memory operation and the two chip select control inputs CS_1, $\overline{CS}2$ are of enabling the chip only when it is selected by the microprocessor.

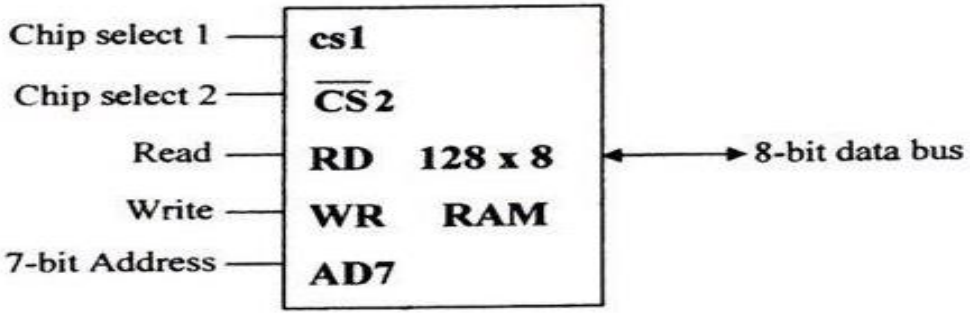


Fig (7) The block diagram of RAM chip

The function table listed in the following show the operations of RAM chip, when the unit is in operation when CS_1 and $\overline{CS2}$ =0 .

CS_1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	x	x	Inhibit	High-impedance
0	1	x	x	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAm
1	0	1	x	Read	Output data from RAm
1	1	x	x	Inhibit	High-inpedace

The Rom chip have 9-address line and the $CS_1=1$ $\overline{CS}2=0$ to operate and do not need to read and write see figure (8).

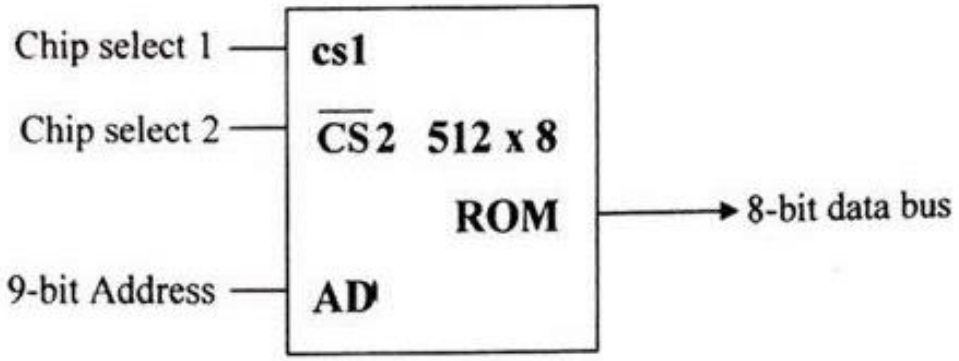
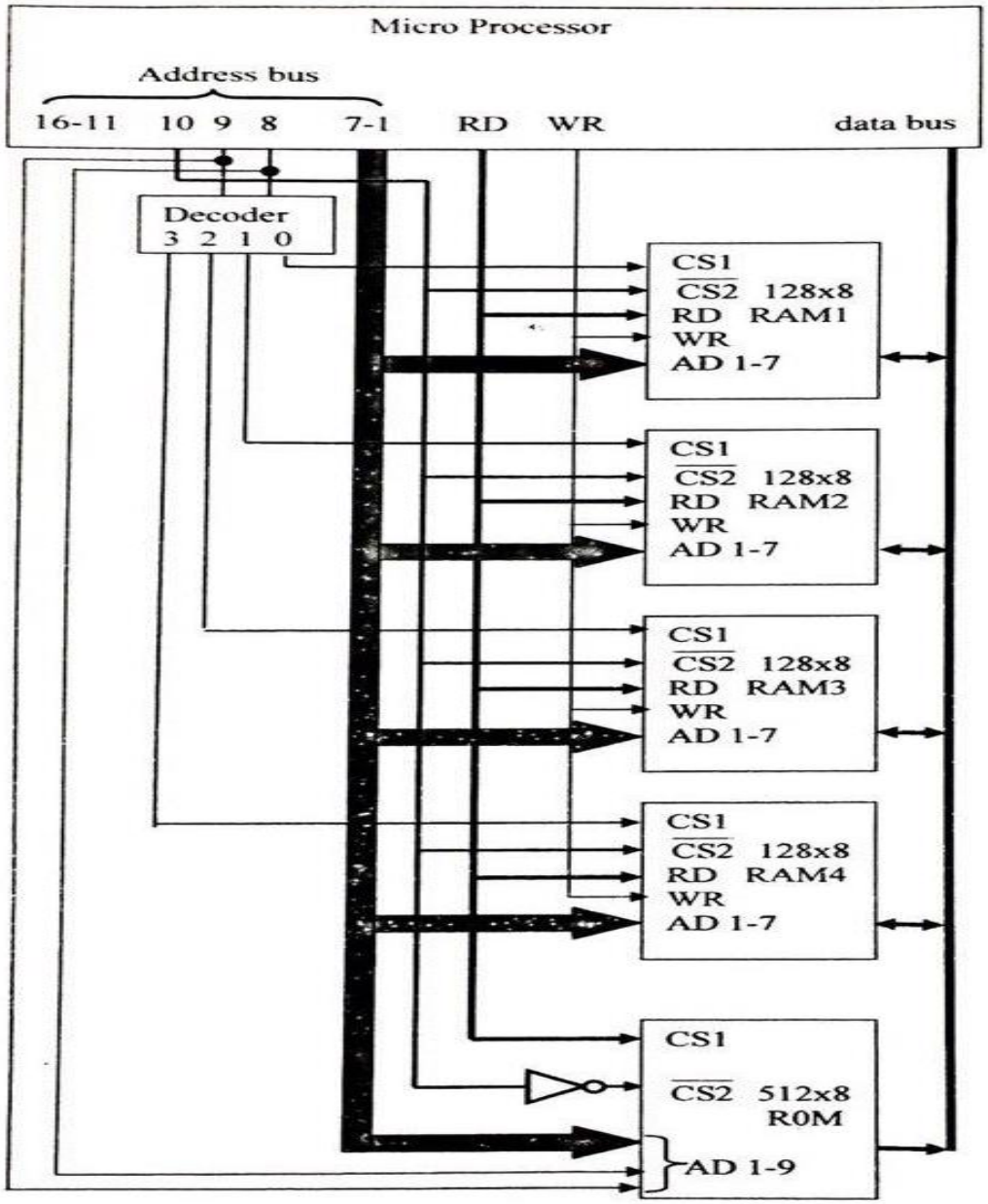


Fig (8) typical ROM chip

Memory Connection to microprocessor:

The connection of memory chip to the microprocessor is shown in Fig. (9) which show a memory capacity of 128 of RAM and 512 bytes of ROM. Each RAM receives the seven low-order bits of the address bus to select one of 128 bytes. The particular RAM chip selected from line 8 and 9 by using 2x4 decoder whose output go to CS1 input in each RAM example when line 8 and 9 equal 00 then the first RAM chip select and when equal 01 the second selected and so on . The selection between RAM and ROM is achieved by bus line (10). The RAM's are selected when the bit in this line is 0 and the ROM is selected when the bit is 1. And the other chip select CS1 in ROM is connected to the RD control line for the ROM chip to be enabled only during a read operation.


Micro Processor
Address bus
16-11
10 9 8
7-1
RD
WR
data bus
Decoder
3 2 1 0
CS1
CS2 128x8
RD RAM1
WR
AD 1-7
CS1
CS2 128x8
RD RAM2
WR
AD 1-7
CS1
CS2 128x8
RD RAM3
WR
AD 1-7
CS1
CS2 128x8
RD RAM4
WR
AD 1-7
CS1
CS2 512x8
ROM
AD 1-9

Thank you …

Q&A

Chapter Four

Chapter Four

Pipeline and Vector Processing

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Today, pipelining is key to making processors fast.

pipeline instruction classically take five steps:

1. Fetch instruction from memory.
2. Read registers while decoding the instruction. The format of MIPS instructions allows reading and decoding to occur simultaneously.
3. Execute the operation or calculate an address.
4. Access an operand in data memory.
5. Write the result into a register.

Hence, the MIPS pipeline we explore in this chapter has five stages. The following example shows that pipelining speeds up instruction execution

EXAMPLE

Single-Cycle versus Pipelined Performance

To make this discussion concrete, let's create a pipeline. In this example, and in the rest of this chapter, we limit our attention to eight instructions: load word (`lw`), store word (`sw`), add (`add`), subtract (`sub`), and (`and`), or (`or`), set-less-than (`slt`), and branch-on-equal (`beq`).

Compare the average time between instructions of a single-cycle implementation, in which all instructions take 1 clock cycle, to a pipelined implementation. The operation times for the major functional units in this example are 200 ps for memory access, 200 ps for ALU operation, and 100 ps for register file read or write. As we said in Chapter 3, in the single-cycle model every instruction takes exactly 1 clock cycle, so the clock cycle must be stretched to accommodate the slowest instruction.

ANSWER

Figure 4.1 shows the time required for each of the eight instructions. The single-cycle design must allow for the slowest instruction—in Figure 4.1 it is `lw`—so the time required for every instruction is 800 ps.

Figure 4.2 compares nonpipelined and pipelined execution of three load word instructions. Thus, the time between the first and fourth instructions in the nonpipelined design is 3×800 ps or 2400 ps.

All the pipeline stages take a single clock cycle, so the clock cycle must be long enough to accommodate the slowest operation. Just as the single-cycle design must take the worst-case clock cycle of 800 ps even though some instructions can be as fast as 500 ps, the pipelined execution clock cycle must have the worst-case clock cycle of 200 ps even though some stages take only 100 ps. Pipelining still offers a fourfold performance improvement: the time between the first and fourth instructions is 3×200 ps or 600 ps.

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add, sub, and, or, slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

FIGURE 4.1 Total time for each instruction calculated from the time for each component.

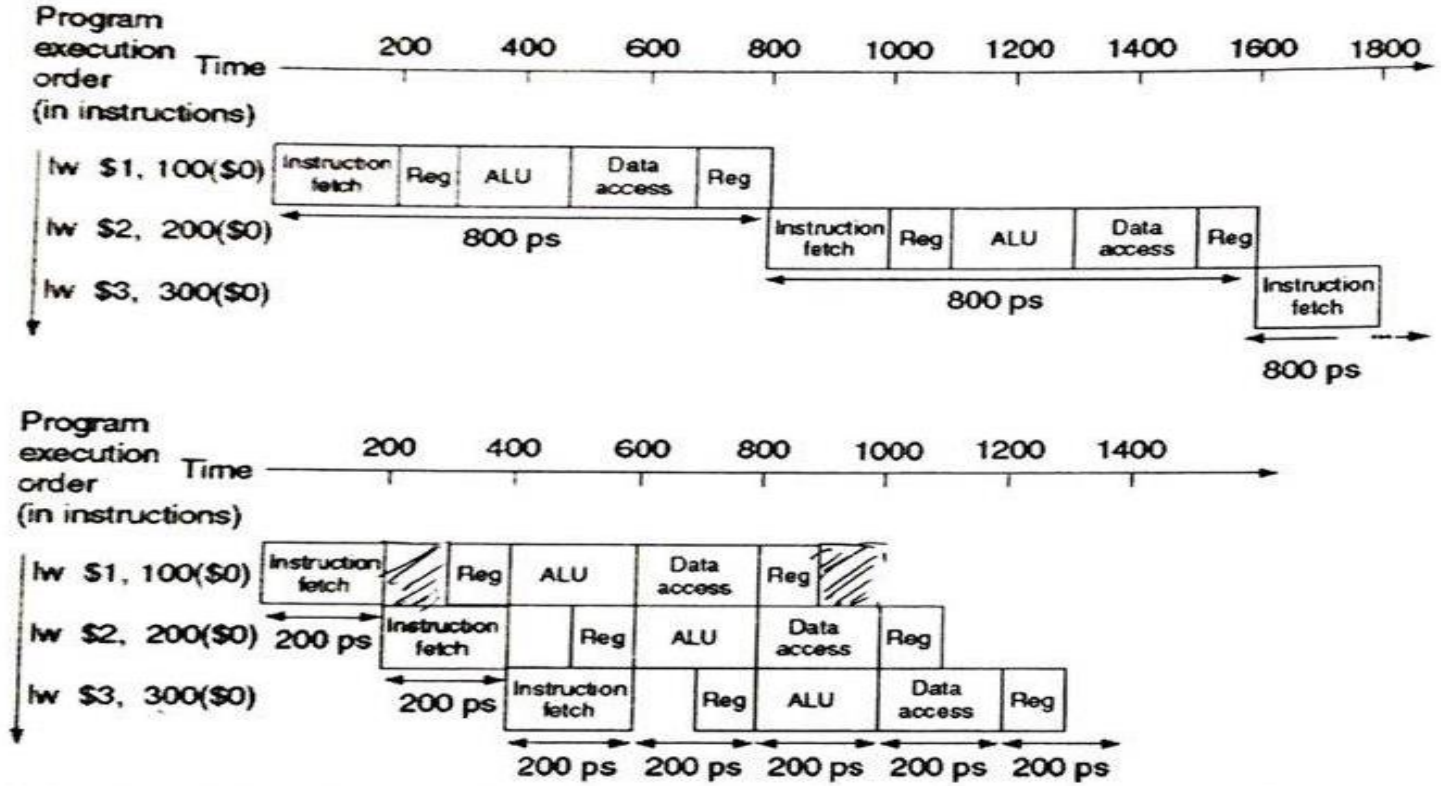


FIGURE 4.2 Single-cycle, nonpipelined execution in top versus pipelined execution in bottom. Both use the same hardware components, whose time is listed in Figure 4.2. In this case we see a fourfold speedup on average time between instructions, from 800 ps down to 200 ps.

. The computer pipeline stage times are limited by the slowest resource, either the ALU operation or the memory access. We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

We can turn the pipelining speedup discussion above into a formula. If the stages are perfectly balanced, then the time between instructions on the pipelined processor—assuming ideal conditions—is equal to

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

Under ideal conditions and with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipe stages; a five-stage pipeline is nearly five times faster.

The formula suggests that a five-stage pipeline should offer nearly a fivefold improvement over the 800 ps nonpipelined time, or a 160 ps clock cycle. The example shows, however, that the stages may be imperfectly balanced. In addition, pipelining involves some overhead, the source of which will be more clear shortly. Thus, the time per instruction in the pipelined processor will exceed the minimum possible, and speedup will be less than the number of pipeline stages.

Moreover, even our claim of fourfold improvement for our example is not reflected in the total execution time for the three instructions: it's 1400 ps versus 2400 ps. Of course, this is because the number of instructions is not large. What would happen if we increased the number of instructions? We could extend the previous figures to 1,000,003 instructions. We would add 1,000,000 instructions in the pipelined example; each instruction adds 200 ps to the total execution time. The total execution time would be 1,000,000 × 200 ps + 1400 ps, or 200,001,400 ps. In the nonpipelined example, we would add 1,0000,000 instructions, each taking 800 ps, so total execution time would be 1,000,000 × 800 ps + 2400 ps, or 800,002,400 ps. Under these ideal conditions, the ratio of total execution times for real programs on nonpipelined to pipelined processors is close to the ratio of times between instructions:

$$\frac{800{,}002{,}400 \text{ ps}}{200{,}001{,}400 \text{ ps}} = 4.00 = \frac{800 \text{ ps}}{200 \text{ ps}}$$

Pipelining improves performance by *increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction*, but instruction throughput is the important metric because real programs execute billions of instructions.

4-2 Instruction Pipeline Design

A stream of instructions can be executed by a pipeline in an overlapped manner. We describe below instruction pipelines for CISC and RISC scalar processors. Topics to be studied include instruction prefetching, data forwarding, hazard avoidance, interlocking for resolving data dependences, dynamic instruction scheduling, and branch handling techniques for improving pipelined processor performance.

4-2-1 Instruction Execution Phases

A typical instruction execution consists of a sequence of operations, including instruction fetch, decode, operand fetch, execute, and write-back phases. These phases are ideal for overlapped execution on a linear pipeline. Each phase may require one or more clock cycles to execute, depending on the instruction type and processor/memory architecture used.

Pipelined Instruction Processing A typical instruction pipeline is depicted in Fig. 4-3 The *fetch stage* (F) fetches instructions from a cache memory, presumably one per cycle.. The *decode stage* (D) reveals the instruction function to be performed and identifies the resources needed. Resources include general-purpose registers, buses, and functional units. The *issue stage* (I) reserves resources. Pipeline control interlocks are maintained at this stage. The operands are also read from registers during the issue stage.

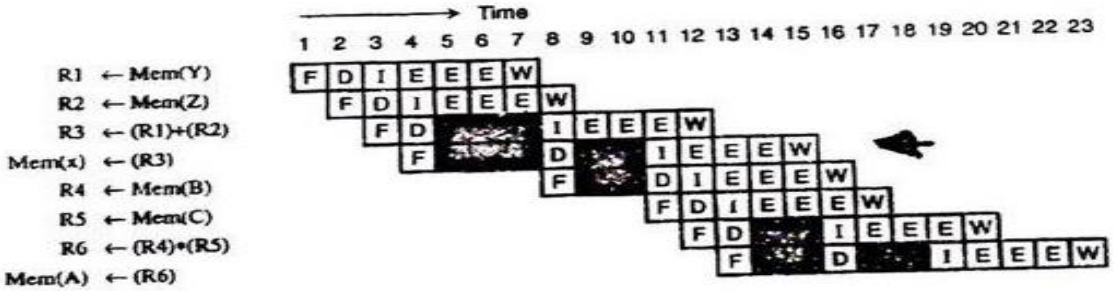
The instructions are executed in one or several *execute stages* (E). Three execute stages are shown in Fig. 4-3 a. The last *writeback stage* (W) is used to write results into the registers. Memory load or store operations are treated as part of execution. Figure 4-3 shows the flow of machine instructions through a typical pipeline. These eight instructions are for pipelined execution of the high-level language statements $X = Y + Z$ and $A = B \times C$. Assume *load* and *store* instructions take four execution clock cycles, while floating-point *add* and *multiply* operations take three cycles.

The above timing assumptions represent typical values used in a CISC processor. In many RISC processors, fewer clock cycles are needed. On the other hand, Cray 1 requires 11 cycles for a load and a floating-point addition takes six. With in-order instruction issuing, if an instruction is blocked from issuing due to a data or resource dependence, all instructions following it are blocked.

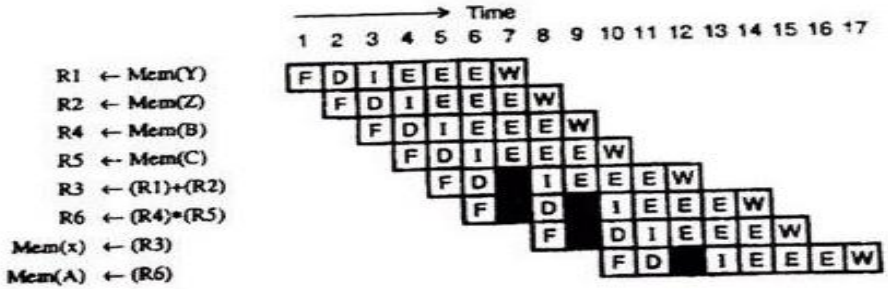
Figure 4-3b illustrates the issue of instructions following the original program order. The shaded boxes correspond to idle cycles when instruction issues are blocked due to resources latency or conflicts or due to data dependences. The first two *load* instructions issue on consecutive cycles. The *add* is dependent on both *loads* and must wait three cycles before the data (Y and Z) are loaded in.

Similarly, the *store* of the sum to memory location X must wait three cycles for the *add* to finish due to a flow dependence. There are similar blockages during the calculation of A. The total time required is 17 clock cycles. This time is measured beginning at cycle 4 when the first instruction starts execution until cycle 20 the last instruction starts execution. This timing measure eliminates the unduly effects of the

(a) A seven-stage instruction pipeline



(b) In-order instruction issuing



(c) Reordered instruction issuing

Figure 4-3 Pipelined execution of X = Y + Z and A = B × C. (Courtesy of James Smith; reprinted with permission from *IEEE Computer*, July 1989)

pipeline "startup" or "draining" delays.

Figure 4-3c shows an improved timing after the instruction issuing order is changed to eliminate unnecessary delays due to dependence. The idea is to issue all four *load* operations in the beginning. Both the *adu* and *multiply* instructions are blocked fewer cycles due to this data prefetching. The reordering should not change the end results. The time required is being reduced to 11 cycles, measured from cycle 4 to cycle 14.

Thank you …

Q&A

Chapter five

Part 1

Chapter 5
Multiprocessors

In this chapter, we study system architectures of multiprocessors and multicomputers. Various cache coherence protocols ·

5.1 Multiprocessor System Interconnects

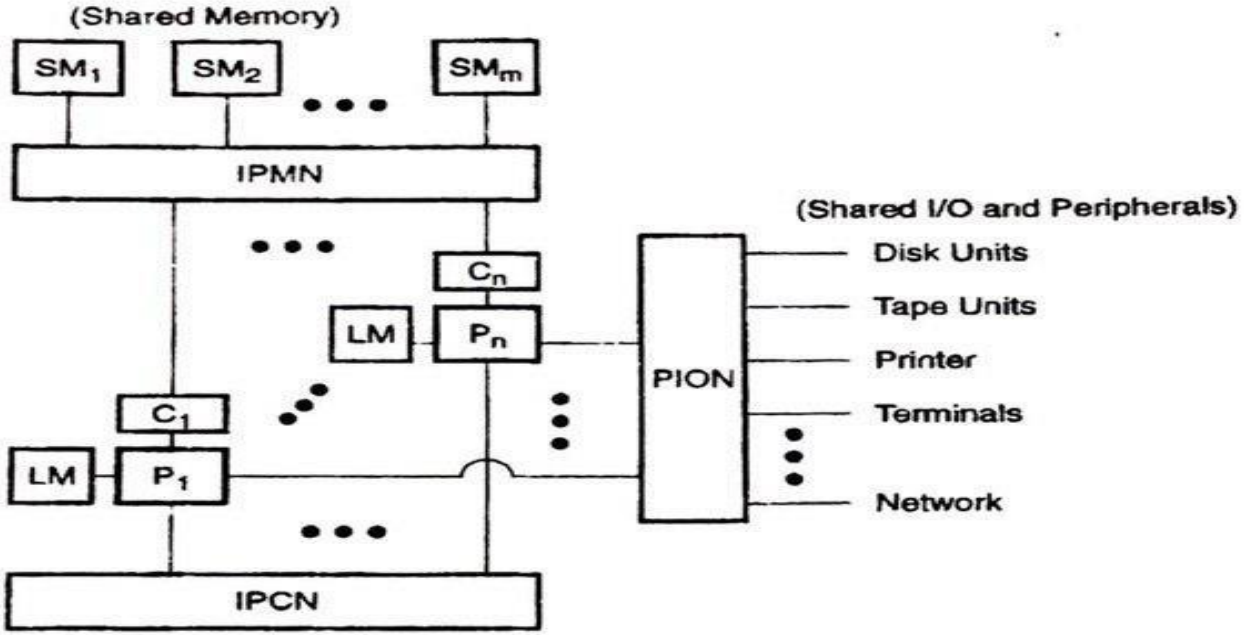
Parallel processing demands the use of efficient system interconnects for fast communication among multiple processors and shared memory, I/O, and peripheral devices. Hierarchical buses, crossbar switches, and multistage networks are often used for this purpose.

A generalized multiprocessor system is depicted in Fig. 5.1.

Each processor P_i is attached to its own local memory and private cache. Multiple processors are connected to shared-memory modules through an interprocessor-memory network (IPMN).

The processors share the access of I/O and peripheral devices through a processor-I/O network (PION). Both IPMN and PION are necessary in a shared-resource multiprocessor. Direct interprocessor communications are supported by an optional interprocessor communication network (IPCN) instead of through the shared memory.

Network Characteristics Each of the above types of networks can be designed with many choices. The choices are based on the topology, timing protocol, switching method, and control strategy. Dynamic networks are used in multiprocessors in which the interconnections are under program control. Timing, switching, and control are



Legends:

IPMN	(Inter-Processor-Memory Network)
PION	(Processor-I/O Network)
IPCN	(Inter-Processor Communication Network)
P	(Processor)
C	(Cache)
SM	(Shared Memory)
LM	(Local Memory)

Figure 5.1 Interconnection structures in a generalized multiprocessor system with local memory, private caches, shared memory, and shared peripherals.

three major operational characteristics of an interconnection network. The timing control can be either *synchronous* or *asynchronous*. Synchronous networks are controlled by a global clock that synchronizes all network activities. Asynchronous networks use handshaking or interlocking mechanisms to coordinate fast and slow devices requesting use of the same network.

A network can transfer data using either *circuit switching* or *packet switching*. In circuit switching, once a device is granted a path in the network, it occupies the path for the entire duration of the data transfer. In packet switching, the information is broken into small packets individually competing for a path in the network.

Network control strategy is classified as *centralized* or *distributed*. With centralized control, a global controller receives requests from all devices attached to the network and grants the network access to one or more requesters. In a distributed system, requests are handled by local devices independently.

5.2 Hierarchical Bus Systems

A *bus system* consists of a hierarchy of buses connecting various system and subsystem components in a computer. Each bus is formed with a number of signal, control, and power lines. Different buses are used to perform different interconnection functions.

In general, the hierarchy of bus systems are packaged at different levels as depicted in Fig. 5.2, including local buses on boards, backplane buses, and I/O buses.

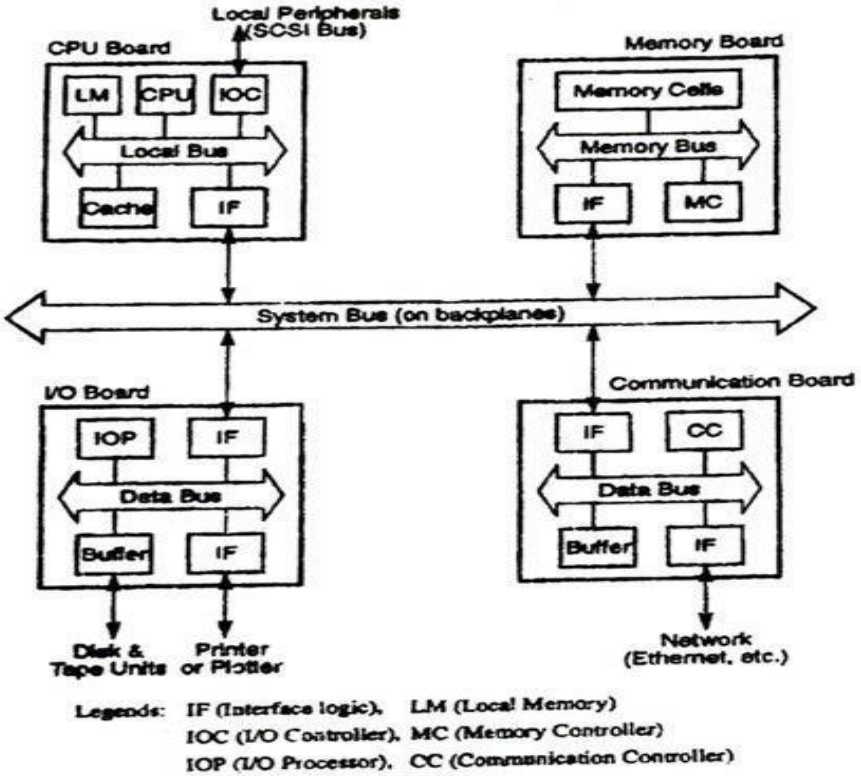


Figure 5.2 Bus systems at board level, backplane level, and I/O level.

Local Bus Buses implemented on *printed-circuit* boards are called *local buses*. On a processor board one often finds a local bus which provides a common communication path among major components (chips) mounted on the board. A memory board uses a *memory bus* to connect the memory with the interface logic.

An I/O board or network interface board uses a *data bus*. Each of these board buses consists of signal and utility lines. With the sharing of the lines by many I/O devices, the layout of these lines may be at different layers of the PC board.

معمارية
محاضره 13

5.3 Cache Coherence and Synchronization Mechanisms

Cache coherence protocols for coping with the multicache inconsistency problem are considered below. Snoopy protocols are designed for bus-connected systems. Directory-based protocols apply to network-connected systems.

5.3.1 The Cache Coherence Problem

In a memory hierarchy for a multiprocessor system, data inconsistency may occur between adjacent levels or within the same level. For example, the cache and main memory may contain inconsistent copies of the same data object. Multiple caches may possess different copies of the same memory block because multiple processors operate asynchronously and independently.

Caches in a multiprocessing environment introduce the *cache coherence problem*. When multiple processors maintain locally cached copies of a unique shared-memory location, any local modification of the location can result in a globally inconsistent view of memory. Cache coherence schemes prevent this problem by maintaining a uniform state for each cached block of data. Cache inconsistencies caused by data sharing, process migration, or I/O are explained below.

Inconsistency in Data Sharing The cache inconsistency problem occurs only when multiple private caches are used. In general, three sources of the problem are identified: *sharing of writable data*, *process migration*, and *I/O activity*. Figure **5.3** illustrates the problems caused by the first two sources. Consider a multiprocessor with two processors, each using a private cache and both sharing the main memory. Let X be a shared data element which has been referenced by both processors. Before update, the three copies of X are consistent.

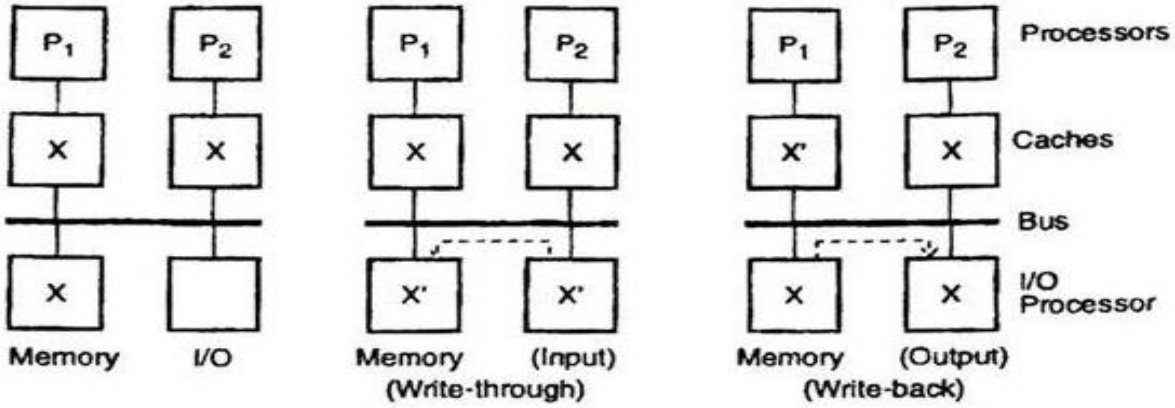
If processor P_1 *writes* new data X' into the cache, the same copy will be written immediately into the shared memory using a *write-through* policy. In this case, inconsistency occurs between the two copies (X' and X) in the two caches (Fig. **5.3** a).

On the other hand, inconsistency may also occur when a *write-back* policy is used, as shown on the right in Fig. **5.3** a. The main memory will be eventually updated when the modified data in the cache are replaced or invalidated.

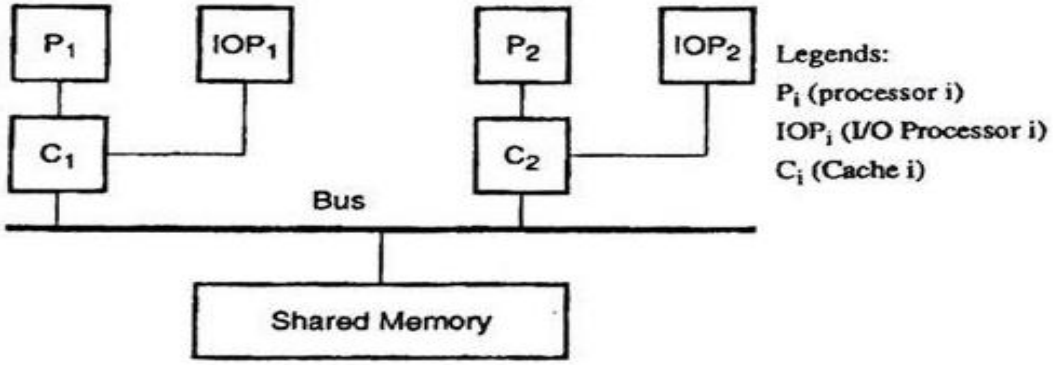
Process Migration and I/O Figure **5.3b** shows the occurrence of inconsistency after a process containing a shared variable X migrates from processor 1 to processor 2 using the write-back cache on the right. In the middle, a process migrates from processor 2 to processor 1 when using write-through caches.

In both cases, inconsistency appears between the two cache copies, labeled X and X'. Special precautions must be exercised to avoid such inconsistencies. A coherence protocol must be established before processes can safely migrate from one processor to another.

Inconsistency problems may occur during I/O operations that bypass the caches.



(a) I/O operations bypassing the cache



(b) A possible solution

Figure 5.4 **Cache inconsistency after an I/O operation and a possible solution.** (Adapted from Dubois, Scheurich, and Briggs, 1988)

transactions. If a bus transaction threatens the consistent state of a locally cached object, the cache controller can take appropriate actions to invalidate the local copy. Protocols using this mechanism to ensure coherence are called *snoopy protocols* because each cache snoops on the transactions of other caches.

On the other hand, scalable multiprocessor systems interconnect processors using short point-to-point wires in direct or multistage networks. Unlike the situation in buses, the bandwidth of these networks increases as more processors are added to the system. However, such networks do not have a convenient snooping mechanism and do not provide an efficient broadcast capability. In such systems, the cache coherence problem can be solved using some variant of directory schemes.

In general, a cache coherence protocol consists of the set of possible states in the local caches, the state in the shared memory, and the state transitions caused by the messages transported through the interconnection network to keep memory coherent. In what follows, we first describe the snoopy protocols and then the directory-based protocols. These protocols rely on software, hardware, or a combination of both for implementation.

possible solution to idling the processor during a long-latency remote memory reference is to switch the processor to another task in the same manner that an operating system will switch contexts when an input/output (I/O) operation is begun. Unfortunately, this requires a highly efficient synchronization mechanism to manage the matching of memory responses with idled, or deferred, tasks.

It was our conclusion that satisfactory solutions to the problems raised for von Neumann architectures can only be had by altering the architecture of the processor itself. Questions raised in this study regarding the near-miss behavior of certain von Neumann multiprocessors (*e.g.*, the Denelcor HEP [22, 30]) led to the belief that dataflow machines and von Neumann machines actually represent two points on a continuum of architectures.

Arvind has suggested that an architecture formed on the principles of *split transaction* I-Structure memory references in a von Neumann framework coupled with data driven rescheduling of suspended instructions in the local memory of each processor would be interesting. Such a machine has the potential of tolerating memory latency and of supporting fine-grained synchronization, and yet (in the strict sense) is neither a von Neumann machine nor a dataflow machine. This suggestion has led me to develop the hybrid architecture presented here. In order to better understand the motivations, the next sections re-examine the strengths and weaknesses of von Neumann and dataflow architectures.

2.2. von Neumann Architectures

Advocates of non-von Neumann architectures (including the author) have argued that the notion of sequential instruction execution is the antithesis of parallel processing. This criticism is actually slightly off the mark. Rather, a von Neumann machine in a multiprocessor configuration does poorly because it fails to provide efficient synchronization support at a low level. Why is this so?

The participants in any one synchronization event require a common ground, a *meeting place*, for the synchronization to happen. This may take the form of a semaphore [9], a register [28, 22], a buffer tag [31], an interrupt level, or any of a number of similar devices. In all cases, one can simply think of the common ground as being the *name* of the resource used (*e.g.*, register number, tag value, etc.). The participants also require a mechanism to trigger synchronization action.

When viewed in this way, it should be clear that the number of simultaneously pending synchronization events is bounded by the size of this name space as well as by the cost of each synchronization operation. More often than not, this name space is tied to a physical resource (*e.g.*, registers) and is therefore quite small, thereby limiting support for low level dynamic synchronization. For most existing von Neumann machines, synchronization mechanisms are inherently larger grain (*e.g.*, interrupts) or involve busy waiting (*e.g.*, the HEP[1] [22, 30]). Therefore, the cost of each event is quite high. Such mechanisms are unsuitable for controlling latency cost. Moreover, since task suspension and resumption typically involve expensive context switching, exploitation of parallelism by decomposing a program into many small, communicating tasks may not actually realize a speed-up.

It is important to observe that these arguments favor the alteration of the basic von Neumann mechanism, and not its total abandonment. For situations where instruction sequencing and data dependence constraints can be worked out at compile time, there is still reason to believe that a von Neumann style sequential (deterministic time order) interpreter provides better control over the machine's behavior than does a dynamic scheduling mechanism and, arguably, better cost-performance. It is *only* in those situations where sequencing cannot be so optimized at compile time, *e.g.*, for long latency operations, that dynamic scheduling and low-level synchronization are called for. One must also keep in mind that,

---

[1]The HEP also exhibited several synchronization namespace problems: the register space was too small (2K), there was a limit of one outstanding memory request per process, and there was a very serious limit of 128 process status words per processor.

Thank you …

Q&A

Chapter five

Part 2

despite any desire to revolutionize computer architecture, von Neumann machines will continue to be the best understood base upon which to build for many years.

2.3. Dataflow Architectures

The MIT Tagged Token Dataflow Architecture, and other dataflow architectures like it, provide well-integrated synchronization at a very basic level. By using an encoded dataflow graph for program representation, machine instructions become self-sequencing. One strength of the TTDA is that each datum carries its own context identifying information. By this mechanism, program parallelism can be easily traded for latency because there is no additional cost above and beyond this basic mechanism for switching contexts on a per-instruction basis.

However, it is clear that not all of the distinguishing characteristics of the TTDA contribute towards efficient toleration of latency and synchronization costs. One very sound criticism is that intra-procedure communication is unnecessarily general. Intuitively, it should not be necessary to create and match tokens for scheduling *every* instruction within the body of a procedure - some scheduling can certainly be done by the compiler. In a dataflow machine, however, data driven scheduling is *de rigueur*. This implies, for instance, that the time to execute the instructions in a graph's critical path is the product of the critical path length and the pipeline depth. One is left to wonder if it might not be possible, even desirable, to optimize this by performing the *necessary* synchronization explicitly, and relying on more traditional (read: *well-understood*) mechanisms for instruction sequencing in the remainder of the cases. The uncertainties in this argument are the fraction of time wherein synchronization is necessary, and the complexity of the mechanisms required.

3. Synthesis

A simple view is that von Neumann and dataflow machines are not, in fact, orthogonal but rather sit at opposite ends of a spectrum of architectures. One might speculate that there are families of machines along this spectrum which trade instruction scheduling simplicity for better low level synchronization. One might further speculate that for some figure of architectural merit, taking into account hardware complexity, instruction scheduling flexibility, and synchronization support, that there exists some optimum point between the two extremes, *i.e.*, a hybrid architecture which synergistically combines features of von Neumann and Dataflow.

Starting with the observation that the costs associated with dataflow instruction sequencing in many instances are excessive, others have suggested that dataflow ideas should be used only at the interprocedural level [23] thereby avoiding dataflow inefficiencies while seemingly retaining certain advantages. This view is almost correct, but ignores the importance of the fundamental issues discussed above. Restricting architectures to this "macro dataflow" concept would amount to giving up what is possibly a dataflow machine's biggest feature - the ability to context switch efficiently at a low level to cover memory latency.

Given this, one is led to ask the following question: *what mechanisms at the hardware level are* essential *for tolerating latency and synchronization costs?* Based on various studies of parallel machines [2, 7, 12, 22] the following conclusions are drawn:

- In general, on a machine capable of supporting multiple simultaneous threads of computation, executing programs expressed as a total ordering of instructions will incur more latency cost than will executing a logically equivalent partial ordering of the same instructions. In fact, for a class of programming languages which are *non-sequential* [33], expressing programs as a partial ordering is a necessary condition for avoiding deadlock. It is assumed, therefore, that the machine language must be able to express partial ordering.

- In any multiprocessor architecture, certain instructions will take an unbounded amount of time to complete (*e.g.*, those involving communication). Such operations can be either atomic, single phase operations or split, multiphase operations[2]. Multiphase processing will always minimize latency cost over single phase processing because the potential exists for covering processor idle time. Based on the frequency of the occurrence of such long latency operations [2] in all but the most trivial parallel computations, efficient multiphase operation requires specific hardware mechanisms [3, 12]. Multiphase instructions are commonly referred to as *split transactions*.

The remainder of this section describes a new, hybrid architecture along with its instruction set and programming model. The architecture can be viewed as either an evolution of dataflow architectures in the direction of more explicit (*i.e.*, compiler directed) control over instruction execution order, or as an evolution of von Neumann machines in the direction of better hardware support for synchronization and better tolerance of long latency operations. The study of this architecture will focus on the frequency of unavoidable run-time synchronization and, therefore, the applicability of compiler-directed control over instruction scheduling in a general-purpose multiprocessor.

3.1. Scheduling Quanta

The central idea of this new architecture involves some reconsideration of the basic unit of work in both dataflow and von Neumann architectures. The unit of parallel computation in a von Neumann machine is the *task*. Inter-task synchronization is typically expensive when it relies on software-implemented mechanisms. Such cost favors large tasks which synchronize infrequently. Within a task, synchronization of *producer* and *consumer* instructions is entirely implicit in the ordering of instructions. Between tasks, barrier synchronization is done explicitly with guards, semaphores, or some other similar mechanism. Context switching is usually done when necessary at the synchronization points, so an important performance metric is the *run length*, or number of instructions between synchronization operations. During such a run, instructions from the same context can enter the pipe at each pipe beat. This kind of locality can often be exploited at the hardware level; however, increasing the locality may imply a loss of parallelism.

This is in sharp contrast to the dataflow model where the basic unit of parallel computation is the instruction. Inter-task (*i.e.*, inter-instruction) synchronization is performed implicitly by the hardware; the single instruction "task" is not awakened until its operands are available. Context switching can and does occur at each pipe beat; any instruction n' which is enabled as a result of the completion of instruction n may not enter the pipeline for a number of cycles equal to the pipeline depth. The intervening cycles must be filled by instructions from another thread of execution, possibly but not necessarily from a related context. Not surprisingly, this model is highly parallel, but the parallelism comes at the expense of some lost locality.

3.1.1. Repartitioning Dataflow Graphs

Consider a graph for a simple code block (Figure 3-1). Note that there is some potential parallelism (lack of interdependence between instructions) in this graph. For example, instructions I1 and I5 do not depend on one another. They depend only on the availability of the values a, b, and c.

More pertinent to this discussion are the instructions of the graph which *directly* depend on one another. Instructions I2 and I3, for example, have an interesting dependence. Having executed I2, it is known from the graph that instruction I3 can be executed because it only depends on I2 and a compile time

---

[2] A *multiphase* operation is one which can be divided into parts which seperately *initiate* the operation and later *synchronize* prior to using the result.

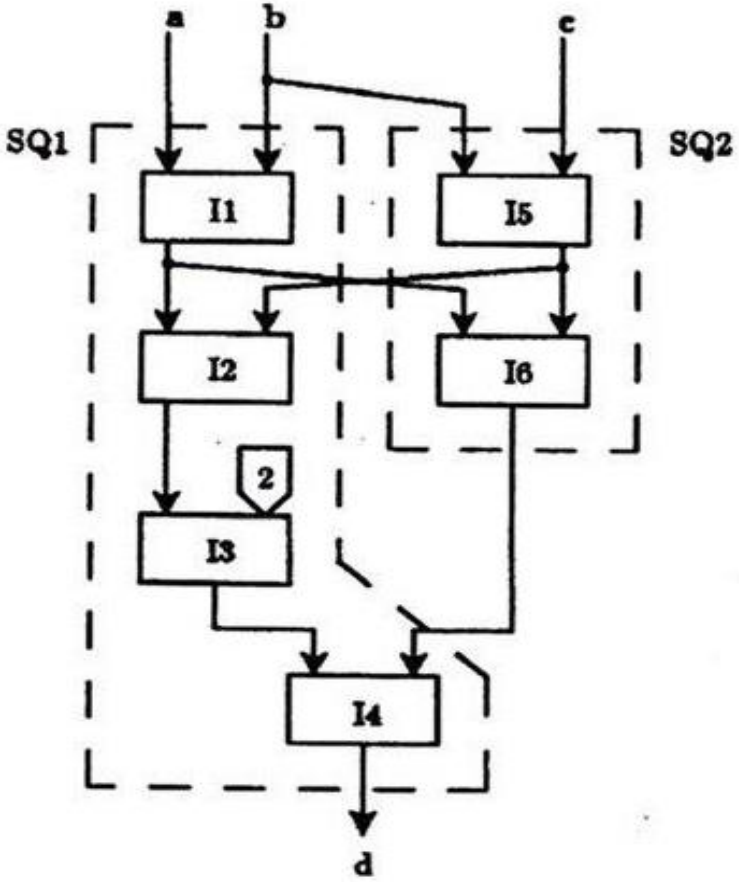


Figure 3-1: A Sample Dataflow Graph

constant. In some sense, then, the pair (I2 , I3) form a new instruction which has the same input and output characteristics as any other instruction, and which has similar synchronization requirements.

There has been some suggestion within the dataflow community that such aggregation be exploited, if only to improve performance. There is a danger in doing this by altering the machine instruction set, because any statistically beneficial aggregation will have been highly dependent on compilation and code generation techniques used while collecting said statistics. That is, the choice of aggregated instructions may vary as improvements are made to the compiler(s). This suggests that the issues of synchronization should be separated from the issues of opcode semantics.

A slightly more sophisticated view is to permit the compiler to aggregate an *arbitrary* collection of instructions according to any criterion of optimality into a unit of schedulability. Each such unit is called a *scheduling quantum*, or *SQ*. Their size, inter-SQ dependences, and content are determined at compile time. In the Figure, two SQ's are shown, but many other aggregations are possible.

3.1.2. Partitioning Strategies

Although the present discussion is oriented toward machine architectures, it is illuminating to look briefly at methods of partitioning programs expressed as graphs into SQ's, partly to lend credibility to the approach, and partly to better understand the relationship between the static and dynamic scheduling requirements of programs. Starting with a dataflow graph, partitioning may be done in a number of ways Issues of concern include:

- **Maximization of exploitable parallelism:** Poor partitioning can obscure inter-procedural and inter-iteration parallelism. The desire to aggregate instructions does not imply any inter-

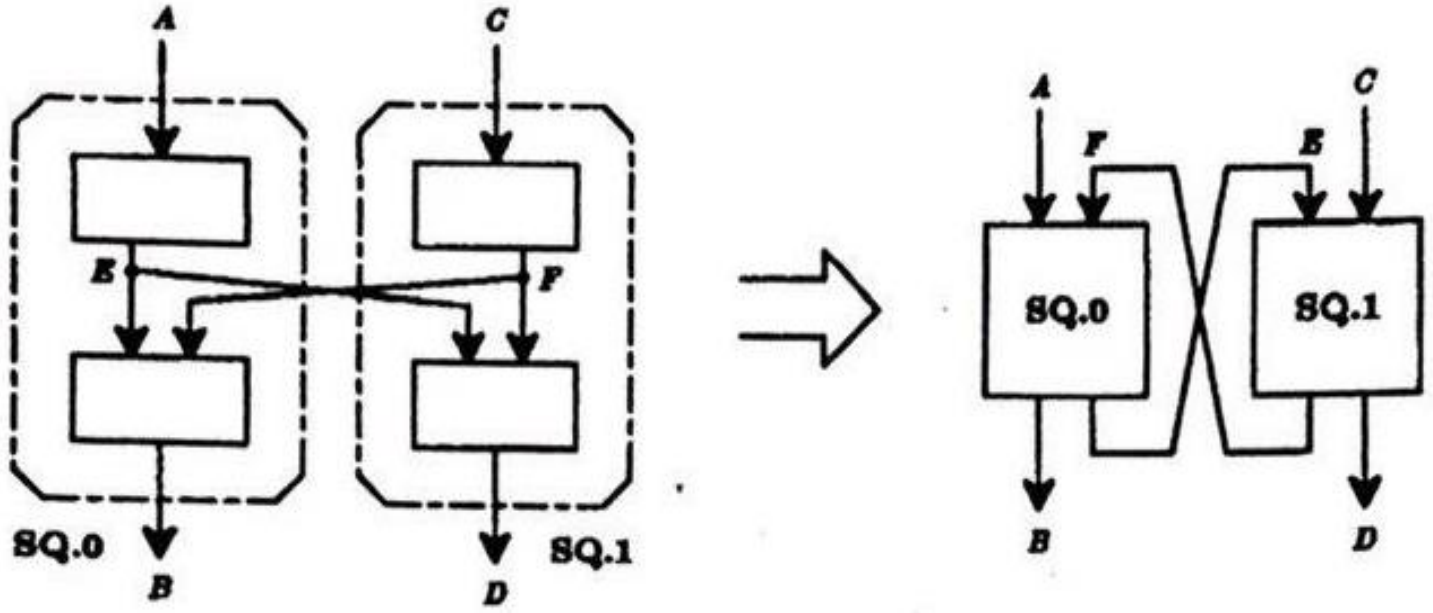


Figure 3-2: Partitioning which Leads to a Static Cycle

est in restricting or limiting parallelism - in fact, those cases where instructions may be grouped into SQ's are quite often places where there is little or no easily exploitable parallelism.

- **Maximization of run length:** Longer SQ's will ultimately lead to longer intervals between context switches (run length). Coupled with proper runtime support for suspension and resumption, this can lead to increased locality. Run lengths which are long compared to the pipeline depth have a positive effect on shortening critical path time. Short run lengths (frequent instruction aborts due to suspension of a frame reference) tend to bubble the pipeline.

- **Minimization of explicit synchronization:** Each arc which crosses SQ boundaries will require dynamic synchronization. Since synchronization operations are pure overhead[3], it is desirable to minimize them.

- **Deadlock avoidance:** Non-sequentiality implies that instruction execution order cannot be made independent of program inputs or, said another way, instruction execution order cannot be determined *a priori*. It is necessary to understand where this dynamic ordering behavior will manifest itself in the generated code. Such dynamic ordering must be viewed as a constraint on partitioning since two instructions whose execution order is dynamically determined cannot be statically scheduled in a single SQ.
- **Maximization of machine utilization:** Given a set of costs for instruction execution, context switching, synchronization, and operand access, partitions can be compared on the basis of how well they "keep the pipeline full." This metric is fairly machine specific and is in that sense less general than those previously described but no less important.

Extant partitioning algorithms [6, 13, 21] can be classified as *depth-first* or *breadth-first*. Depth-first algorithms [6] partition by choosing a path from an input to an output of a graph and making it into an SQ, removing the corresponding instructions from the graph in the process. The algorithm is repeated

---

[3]Coming from a von Neumann uniprocessor mind set where explicit synchronization is virtually unheard of except in situations which require multitasking, it is natural to view synchronization in this way. Coming from the dataflow world where synchronization is unavoidable in every instruction execution, and where there is no opportunity to "optimize it out," it is also reasonable to view explicit synchronization instructions as overhead. In a later section, these perspectives are reconciled with the view that explicit synchronization instructions are both necessary and, in some sense, beneficial.

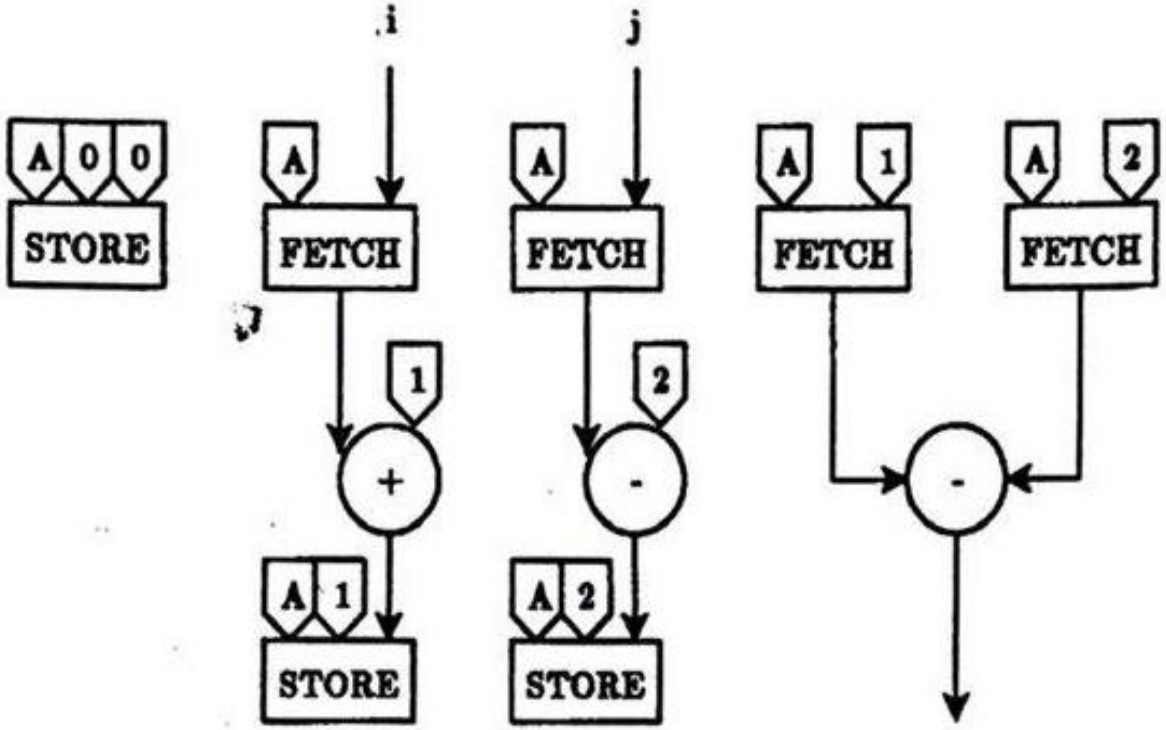


Figure 3-3: Program Graph Fragment

until no instructions remain unpartitioned. Such partitionings tend to be best at minimizing critical path time and rely heavily on pipeline bypassing since, by definition, instruction *n* depends directly on instruction *n-1*. Breadth-first algorithms [13, 21] tend to aggregate instructions which have similar input dependences but only weak mutual dependences. The *method of dependence sets* as presented in [21] is discussed in the next section.

3.1.3. The Method of Dependence Sets

In order to guarantee liveness of the partitioned graph, it is essential no cycle be introduced which cannot be resolved. Such cycles can be either static or dynamic.

> **Definition 1:** An *unresolvable static cycle* is a directed cycle of SQ's in a partitioned dataflow graph for which no schedule of SQ executions can terminate.

An example of how partitioning can give rise to a static cycle is shown in Figure 3-2. It can be shown [21] that a graph interpretation rule which includes *suspension* and *resumption*, *i.e.*, partial execution of SQ's, is a sufficient condition for preventing such static cycles from becoming unresolvable. Sarkar and Hennessy [29] avoid the unresolvability issue entirely by imposing a *convexity constraint* on the partitioning - static cycles can therefore never arise.

A much harder problem is that of preventing unresolvable dynamic cycles. Dynamic cycles arise due to the *implicit* arcs between **STORE** and **FETCH** instructions which refer to identical elements. Such arcs are implicit because they are generally input-dependent.

> **Definition 2:** An *unresolvable dynamic cycle* is a directed cycle of SQ's in a partitioned dataflow graph, augmented with all possible input-specific dynamic arcs, for which no schedule of of SQ executions can terminate.

It is necessary to constrain partitioning such that unresolvable dynamic cycles provably cannot arise for any possible set of program inputs. An example will make this clearer. Consider the following Id program fragment:

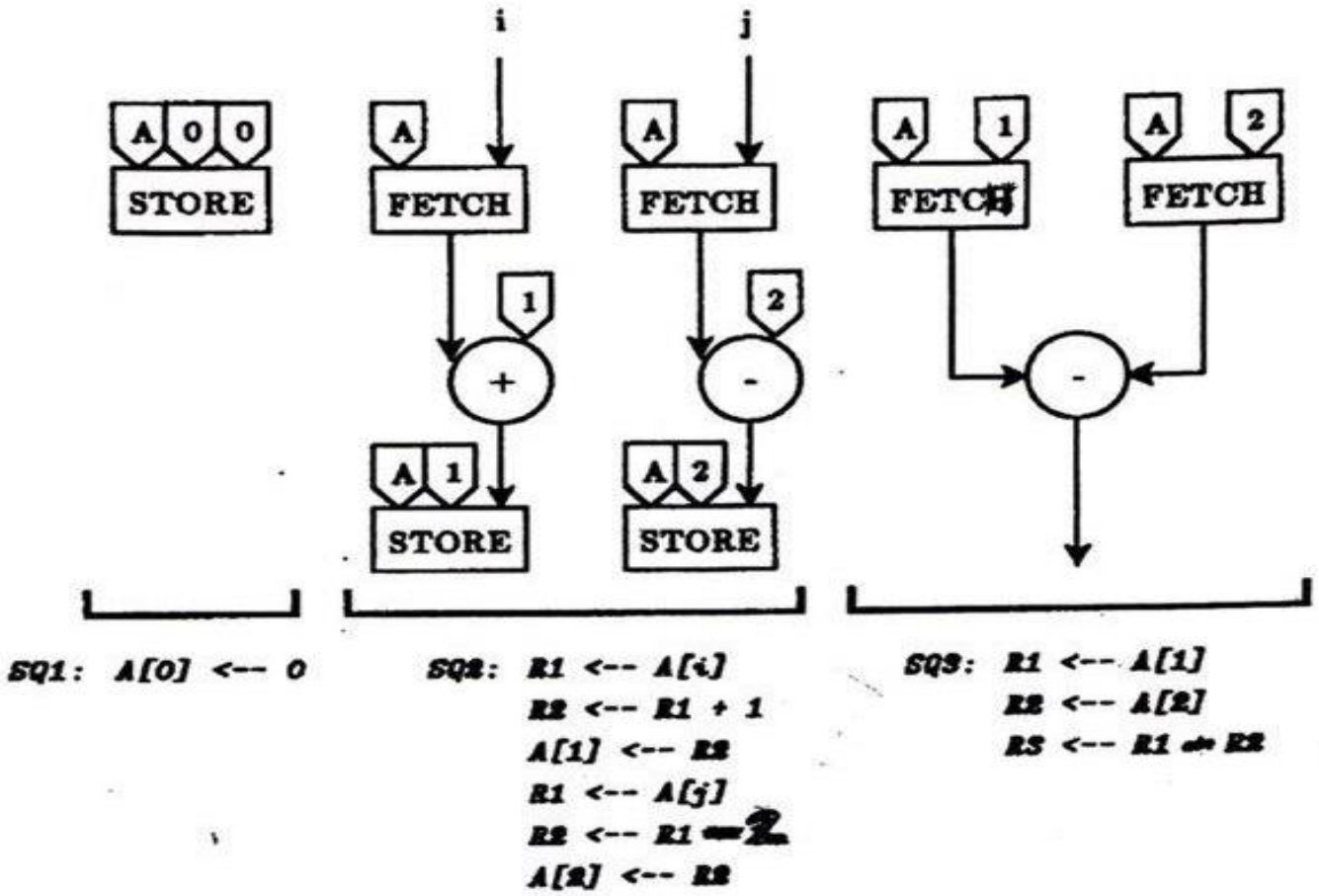


Figure 3-4: Partitioning which Leads to Deadlock

```
{   a = vector (0,2);
 a[0] = 0;
 a[1] = a[i] + 1;
 a[2] = a[j] - 2;

 in a[1] - a[2]}
```

and its associated graph[4,5] in Figure 3-3. Such a graph *would* terminate under a dataflow instruction execution rule. However, without exercising some care, partitioning this graph into SQ's can lead to deadlock. Putting all of these instructions into a single partition won't work, nor will a partitioning such as that shown in Figure 3-4. Such partitionings result in code which can never terminate, despite the adherence to static dependences in deriving the individual SQ schedules.

The problem, of course, is that the actual instruction execution order in the dataflow case depends on the indices used in the structure operations, where no such dependence is allowed in the partitioned case. Figure 3-5 shows two instruction execution orderings which must be possible in any correctly compiled

---

[4]The descriptor for vector *A* is depicted as a constant to simplify the drawings. This is done without loss of generality.

[5]In the sequel, it is assumed that global storage is read by multiphase operations, and that the memory controller implements I-Structure-like synchronization [18]. In that sense, **FETCH** and **STORE** behave as **I-FETCH** and **I-STORE**.

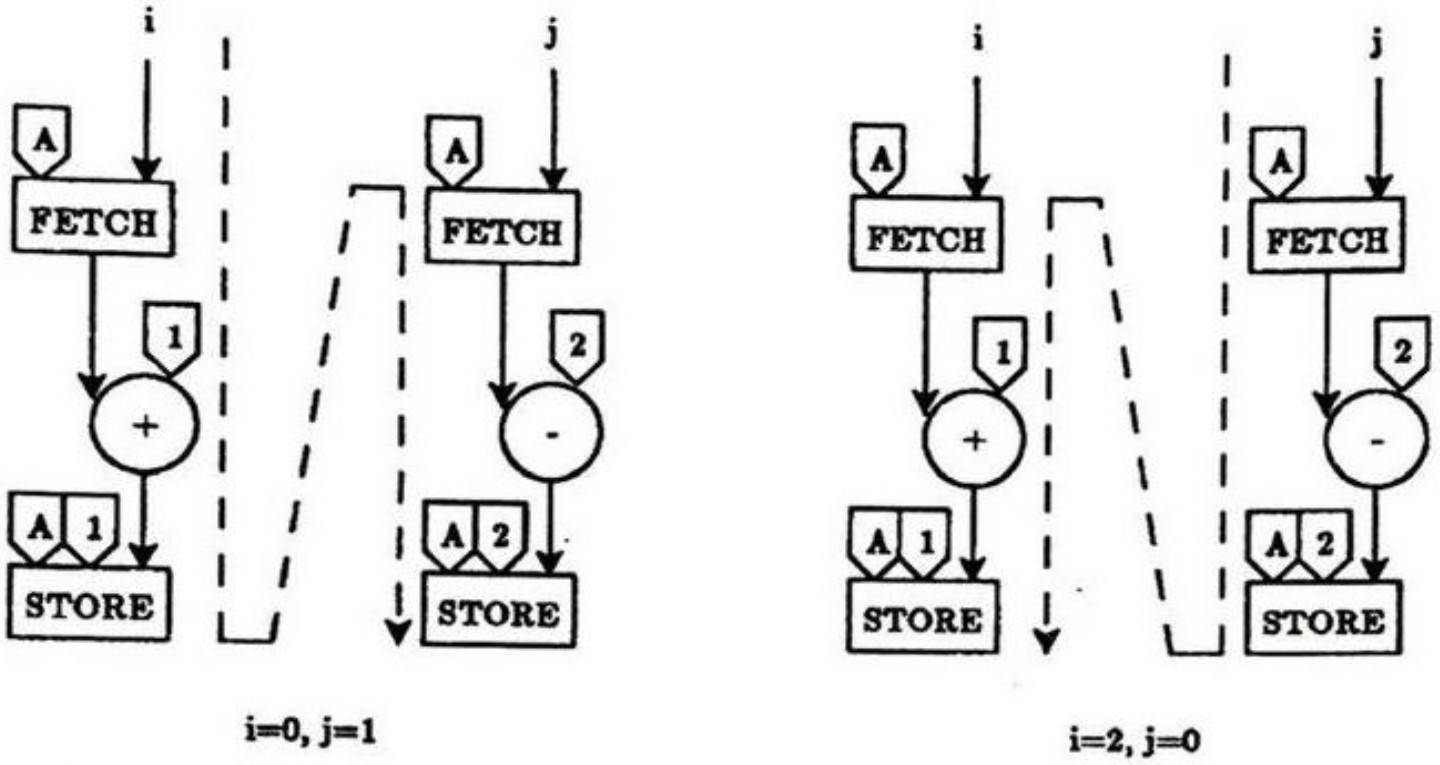


Figure 3-5: Input Dependent Execution Order

version of this program. These orderings demonstrate the dynamic dependences between **STORE**s and **FETCH**es. If these dependences were fixed, and if it were possible to determine them at compile time, SQ partitioning to avoid deadlock would be straightforward. Since this is not the case, the problem is one of developing a safe partitioning strategy which is insensitive to the arrangement of dynamic arcs. One approach is to make each partition exactly one instruction long, *i.e.*, the dataflow method. This, of course, is at odds with the desire to exploit static scheduling.

Another method is to give names to *sets of dependences*. The following definitions are in order:

Definition 3: A ***FETCH-like output*** of an instruction is one which is associated with a dynamic dependence. An instruction itself is **FETCH**-like if at least one of its outputs is **FETCH**-like, implying that the instruction initiates a split transaction (long latency) operation.

Definition 4: The *input dependence set* for an instruction in a well-connected graph [32] is the union of the output dependence sets of all instructions from which it receives input. The input dependence set of the root instruction is defined as $\{ \alpha \}$.

Definition 5: The *output dependence set* for a given output of an instruction is either the instruction's input dependence set if the output is not **FETCH**-like, or the union of the instruction's input dependence set with a singleton set which uniquely names the given output if it is.

Note that it is a **FETCH**-like instruction's *output*, and not the instruction itself, with which is associated a change of dependence set. The intuition is that **FETCH**-like instructions themselves can never suspend while waiting for their output. Rather, the instructions which receive the **FETCH**-like instruction's output are the ones which will suspend. Figure 3-6 makes this clearer. A **FETCH**-like instruction can be viewed as gating the value of a **STORE**-like operation; the dynamic arc terminates on the virtual gate. It has the effect of suspending the "+" instruction until both the **STORE** and the **FETCH** have completed.

Applying the definitions to the graph in Figure 3-3 and using β, γ, and δ (in that order) for unique names results in the following assignments of input dependence sets to instructions (assume that vector *A* and the indices *i,j* are derived from the root with dependence set $\{\alpha\}$):

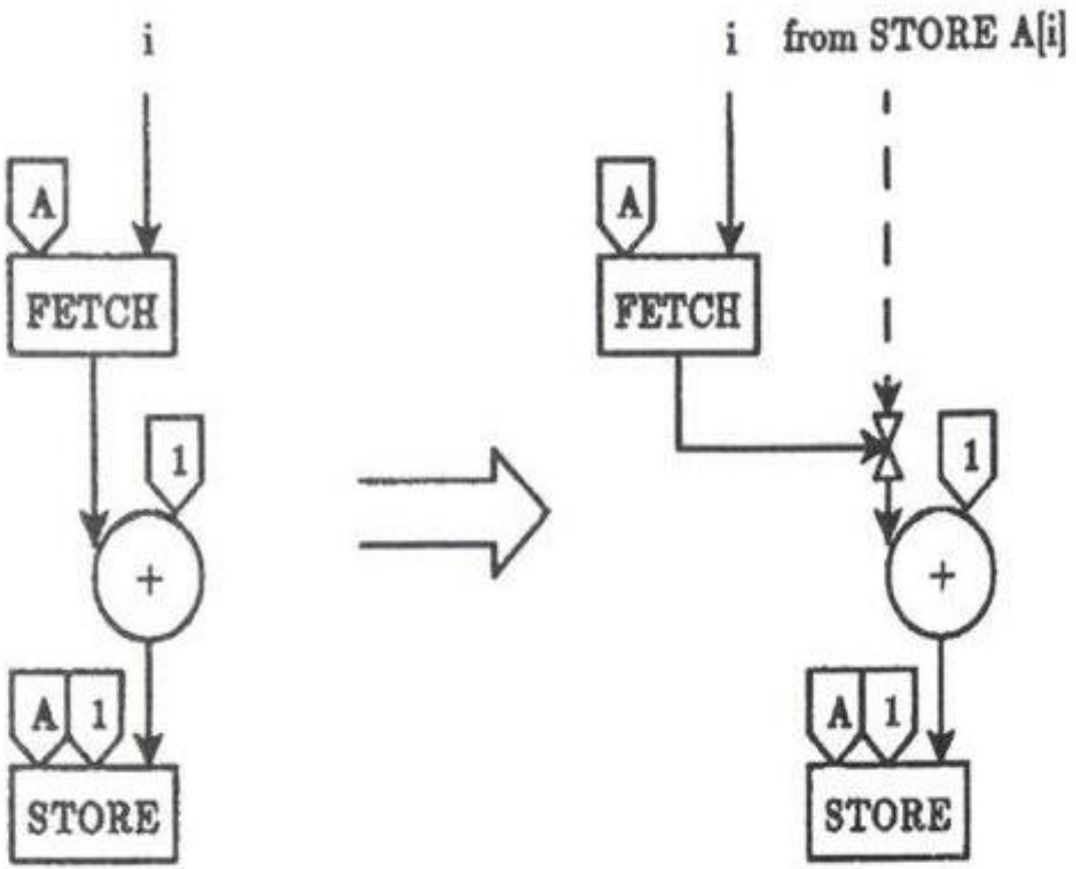


Figure 3-6: Gating Effect of FETCH-like Instructions

Instruction	Input Dependence set
STORE(0)	{ α }
FETCH(i)	{ α }
FETCH(j)	{ α }
FETCH(1)	{ α }
FETCH(2)	{ α }
+1	{ α β }
STORE(1)	{ α β }
-2	{ α γ }
STORE(2)	{ α γ }
-	{ α δ }

The assignment of instructions to SQ's is now straightforward: an SQ is associated with each unique dependence set. Instructions are assigned to the SQ corresponding to their input dependence set in an order corresponding to their topological ordering in the unpartitioned graph. Since each distinct combination of dynamic arcs denotes a single SQ, dynamic scheduling can change to match the dynamic dependences. The correctly partitioned graph is shown in Figure 3-7. The determination of synchronization points is also straightforward: each dependence (arc) which crosses SQ boundaries must be explicitly synchronized by the consumer, or *sink*, SQ. Consumers in the same SQ as the instruction producing a value need not perform synchronization - it is implicit in the static scheduling of instructions within the SQ.

In [21], the deadlock-avoidance property of this algorithm is proved. Moreover, if procedure calls are compiled as **FETCH**-like operations, the method of dependence sets naturally allows inter-procedural parallelism. A simple extension to *k-bounded loops* [10] also allows inter-iteration parallelism. Run length, explicit synchronization, and machine utilization properties of this algorithm are studied in a later section.

Thank you …

Q&A

Chapter five

Part 3

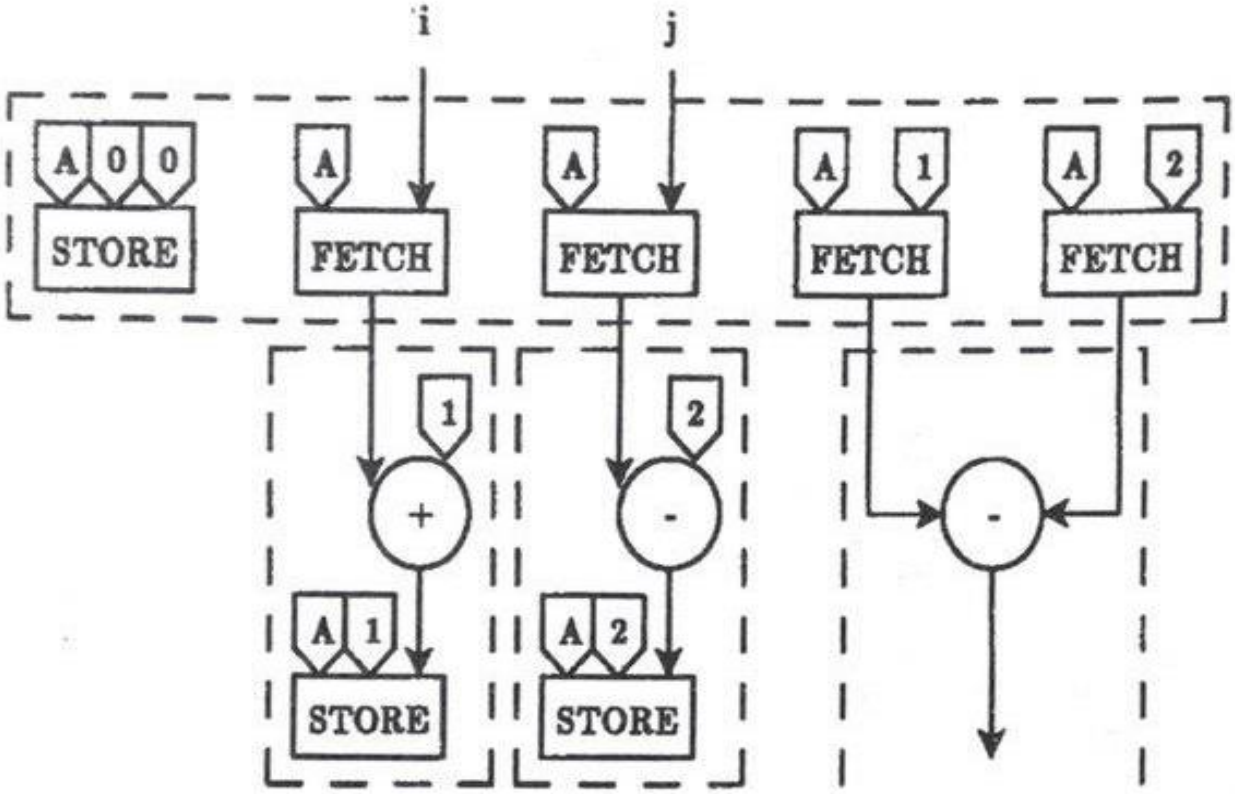


Figure 3-7: Properly Partitioned Graph

3.2. Parallel Machine Language

Let's review the essential conclusions so far. Latency and synchronization have been shown to be fundamental issues in the development of scalable, general purpose multiprocessors, and the issues seem related in fairly incestuous ways. Basic changes to traditional architecture are necessary for dealing with them. One such change is that the execution time for any given instruction must be independent of latency (giving rise to split transactions). A second change is that synchronization mandates hardware support; each synchronization event requires a unique name. The name space is necessarily large, and name management must be efficient. To this end, a compiler should generate code which calls for synchronization when and only when it is necessary. A natural approach is to extend instruction sets to express the concepts of *both* implicit and explicit synchronization. Such an instruction set, which captures the notions of bounded instruction execution time, a large synchronization name space, and means of trading off between explicit and implicit synchronization is called a *parallel machine language* (PML).

It has been shown that adding partitioning to a dataflow graph is tractable. Doing so moves dataflow graphs into the realm of parallel machine languages. The question remains of how to organize a machine to efficiently implement a PML. One method would be to start with the explicitly-synchronized dataflow paradigm and to augment it with facilities for compiler-directed instruction scheduling (*e.g.*, *Monsoon* [24]). Another approach, described below, would be to start with the implicitly-synchronized von Neumann paradigm and to augment it with facilities for dynamic instruction scheduling.

3.3. The Hybrid Multiprocessor

The architecture is modeled as an array of n identical processors, connected through a suitable switching

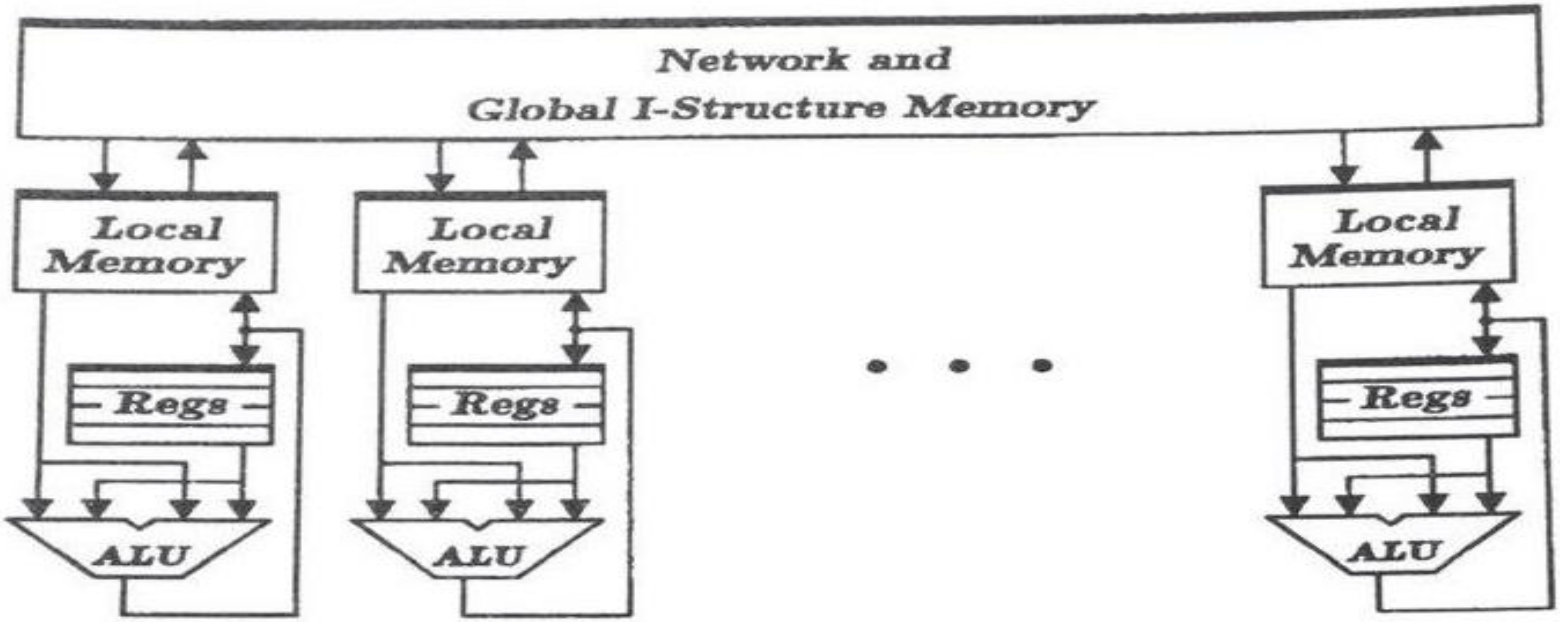


Figure 3-8: The Hybrid Machine

network to a globally addressed I-Structure memory[6]. Each processor is made up of a pipelined datapath, a collection of high speed registers, and a local memory. Instructions are provided which allow movement of data between local and global memories, and between registers and local memories. All interprocessor communications can be thought of as going through *global memory*[7]. The *local memory* is both physically and logically local to a processor. For each invocation of each code block, a *frame* is allocated in the local memory of *exactly one processor* to hold local variables[8]. References to frame slots can be synchronizing or non-synchronizing.

3.3.1. Processor Hardware

The hardware which makes up a hybrid processor is strongly similar to that of a von Neumann machine, but with a few important differences (Figure 3-9). The datapath (ALU, etc.) and registers are conventional. The hardware datatypes are integers, floating point numbers, memory addresses, and the like. The most significant new datatype is the *continuation* which is a tuple of a program counter (PC) and a frame base register (FBR). Logical continuation states are depicted in Figure 3-10 and are encoded by the *location* of the continuation. *Enabled* continuations reside in the Enabled Continuation Queue. The *running* continuation resides in the Active Continuation Register. *Suspended* continuations reside in frame slots. *Uninitiated* and *terminated* continuations are not explicitly represented.

---

[6]The behavior of an I-Structure Storage unit is discussed extensively in [18, 19] and will not be repeated here. It is sufficient to note that all I-Structure references are split transactions and, therefore, never block the processor pipeline. Moreover, one can view the functions of an I-Structure storage as a superset of the functions of a traditional store, *i.e.*, at the hardware level, imperative reads and writes can be performed as easily as I-Structure reads and writes.

[7]This restriction can be relaxed somewhat. It is possible to pass around local memory pointers if they are used for *remote store-in*, *i.e.*, direct forwarding of values from one processor to another. This can be used to great advantage in procedure linkage. The ability to do this is a function of the lifetimes of local memory addresses.

[8]Frame sizes are determined at compile time.

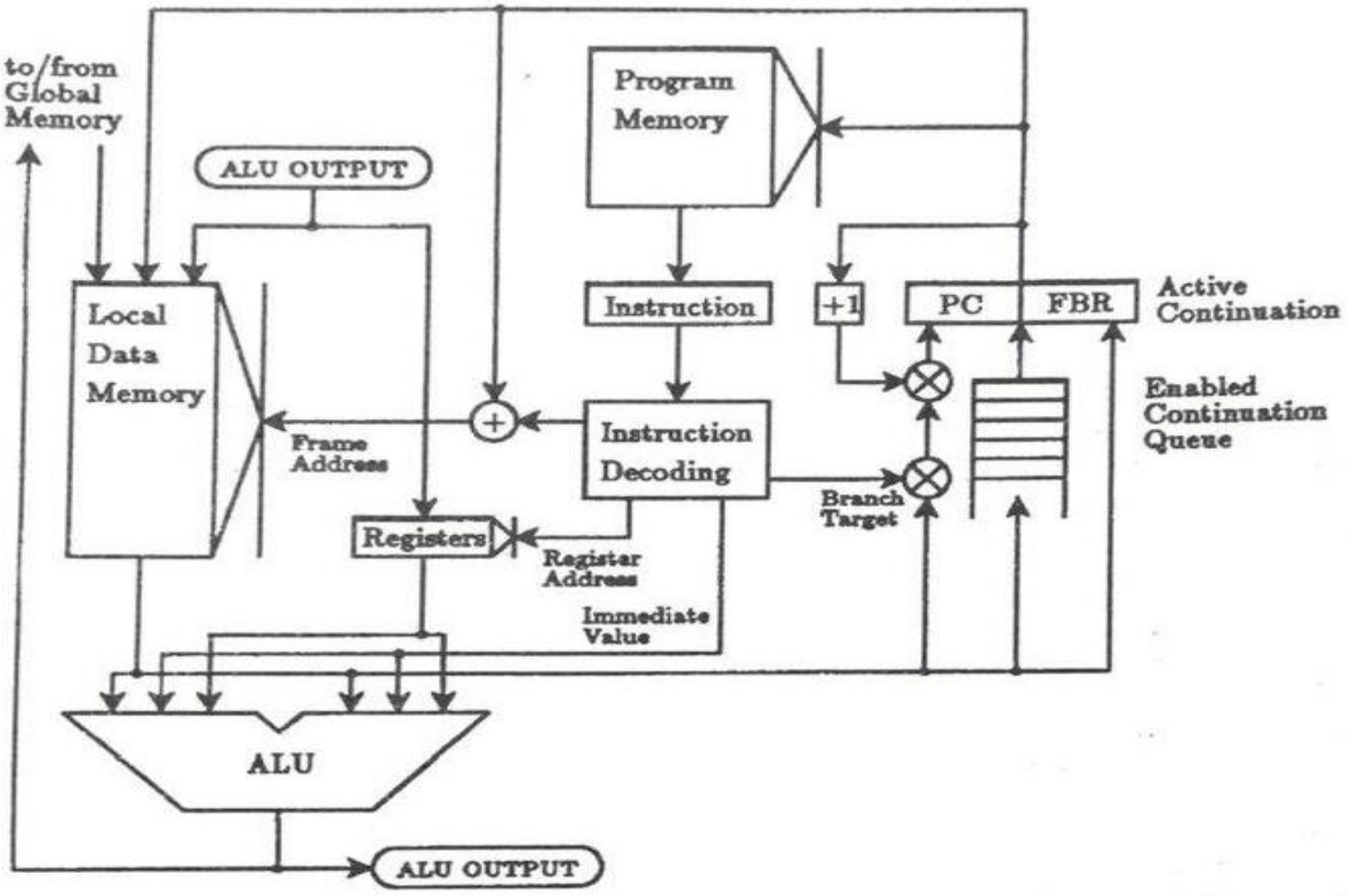


Figure 3-9: The Hybrid Processor

The PC of the running continuation denotes the instruction to be dispatched next. Instructions may make operand references to the registers or to slots in the local data memory. The local memory's behavior is similar to I-Structure storage in that each slot has several presence bits associated with it. A *non-synchronizing* reference to a slot behaves as a normal memory read operation. *Synchronizing* references invoke suspension of the running continuation if the slot being read is marked as EMPTY. Synchronizing reads of a WRITTEN slot behave just the same as nonsynchronizing reads.

The key hardware extension lies in the efficient state-transition management for continuations. Because continuations are word-sized objects, they can be easily fabricated when an SQ is invoked. When the running continuation encounters a blockage (via a synchronizing reference to an empty frame slot), the hardware simply stores the running continuation into the slot, marking it as now containing a continuation. An enabled continuation can then be selected from the queue. Upon satisfaction of the blockage (some other continuation writes into the slot), the suspended continuation is extracted and re-queued. A continuation may be suspended a number of times between initiation and termination. This behavior is reminiscent of the traditional *task* in a demand paged system which, upon encountering a missing memory page, becomes suspended until the page is made available.

Registers may be used freely *between* instructions which make suspensive references to the frame, but since no register saving takes place when a suspension occurs, register contents cannot be considered

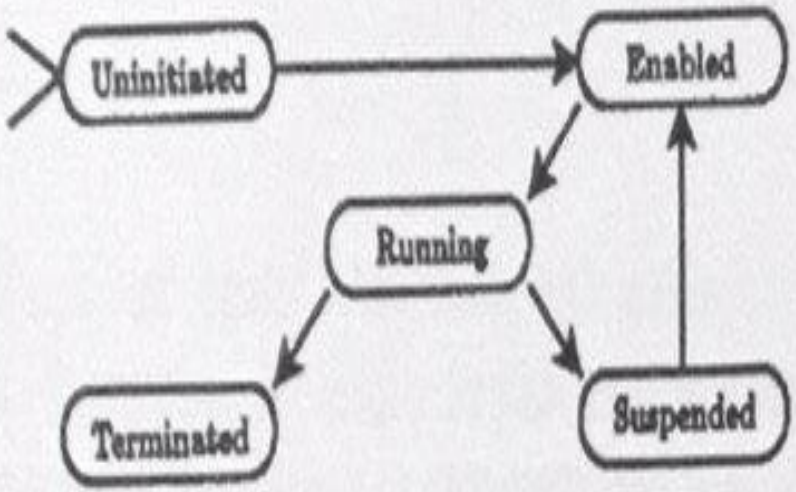


Figure 3-10: Continuation States

valid across potentially suspensive instructions. The cost of a register reference (maximum two per pipeline beat) is less than a local memory reference (maximum one per pipeline beat) which is, in turn, less than a global I-Structure reference (split transaction, possibly with an explicit address arithmetic instruction). Local memory is referenced relative to the FBR in the current continuation. I-Structure address space is global and is shared.

One can imagine a number of implementations of suspension which incur costs ranging from nothing to many tens or hundreds of instructions. In order to keep the implementation from distorting the kinds of code a compiler might generate, it is imperative that the cost of performing a context switch must be exceedingly low - on the order of a single pipe beat. To make this practical and general, context state must be easily saved in a single cycle. Moreover, it must be nearly trivial to create or destroy such context states as SQ's are initiated and terminated. Lastly, the number of such extant contexts cannot be bounded by some small hardware resource. For these reasons, context state *at suspension points* is represented solely by the continuation and its associated frame. By making the continuations no larger than a frame slot, saving and restoring becomes nearly trivial (a single memory reference) compared to schemes wherein registers are also saved[9].

The simple minded *dispatch instructions sequentially until blocked* paradigm works well, and has a very positive effect on locality. Other approaches are possible, however. The high-level goal is to dispatch instructions so as to keep the pipeline full of useful work. *Non-useful* work includes execution of NO-OPs (*i.e.*, pipeline bubbles) and instructions which suspend. Each processor maintains a queue of enabled continuations. One may view each continuation as logically contributing one instruction (the one pointed to by its PC) to the set of *enabled instructions*. At each time step, the processor's instruction dispatcher can freely choose one instruction from among this set. Optimal dispatching of instructions is impossible without foreknowledge of which instructions in the set *will* suspend. However, simple decoding of instructions allows the dispatcher to know if the instruction *cannot* suspend (*e.g.*, those which only reference registers or which make nonsuspensive references to frame memory) or if it *might possibly* suspend. A good strategy, then, is to divide the set of enabled instructions along these lines and to dispatch first from the subset of those which cannot suspend, delaying as long as possible execution of instructions which might suspend (thereby reducing the probability of suspension in many cases)[10].

---

[9]Intuition leads one to believe that such a scheme results in degraded performance in the form of additional memory references. As discussed in [21], this is an oversimplification because frame storage can be cached easily *without* a coherence problem. See Section 5.4 for experience generating code from dataflow graphs shows this strategy to work. More sophisticated code generation techniques can make even better use of such non-saved registers.

[10]This scheme is not ideal, however. Consider the case of having postponed execution of a long-latency but potentially suspensive FETCH. If executing the instruction does not, in fact, cause suspension, delaying it will give up cycles which could otherwise have been used to mask the latency.

3.3.2. Instruction Set

The instruction set is simple and regular in structure, with addressing modes and instruction functions being largely orthogonal. Instructions are readily implemented in a single cycle. The basic addressing modes are IMMEDIATE, REGISTER, FRAME NONSUSPENSIVE, and FRAME SUSPENSIVE. All unary and binary ops for arithmetic and logicals can take immediate, register, or frame slot operands and produce register or frame slot results. The **MOVE** opcode encodes all intra-process data movement. It is capable of moving an immediate, register, or frame slot to a register or frame slot. The **MOVE-REMOTE** opcode initiates movement of a value to a remote (non-local) frame slot. This instruction is used for procedure linkage, and is the only way one procedure can store into another's frame. The **LOAD-FRAME-INDEX** opcode and its variants initiate an indexed read from I-structure memory to the frame. **STORE** and its variants initiate a store to I-Structure memory. The **TEST** and **RESET** opcodes are provided for explicit synchronization and frame slot re-use, respectively. **RESET**s occur within the body of a multiple SQ loop to re-enable synchronization prior to iteration. The **BRANCH** and **BRANCH-FALSE** opcodes do the obvious things, causing the PC in the continuation to be replaced (conditionally in **BRANCH-FALSE**). The **CONTINUE** opcode causes a fork by creating and queueing a new continuation. The corresponding join operation is implemented implicitly through frame slots. A number of other instructions unique to TTDA-style program graphs have been implemented. The simplest are the **CLOSURE** ops which construct and manipulate closures as word-sized objects (rather than memory-bound structures). These are arguably easy to implement in single machine cycles. The remainder are instructions which form manager message packets to allocate and deallocate various resources.

In translating program graphs to machine language, arcs are mapped to frame slots. Slots may be re-used within a code block but it is the responsibility of the compiler to guarantee that all reading of a slot is complete before it is re-written. Synchronizing operand reads are used to implement inter-SQ communication including the synchronization associated with **FETCH**-like instructions. Note that it is the *reader* of the slot which chooses to synchronize or not; it is not a property of the slot itself. Each slot may have multiple readers, some synchronizing and some nonsynchronizing.

Thank you …

Q&A