MODULE-5

Chapter-10 Parallel Programming Models, Languages and Compilers

10.1 Parallel Programming Models

Programming model -> simplified and transparent view of computer hardware/software system.

- Parallel Programming Model are specifically designed for multiprocessors, multicomputer or vector/SIMD computers.

We have 5 programming models-:

1. Shared-Variable Model
2. Message-Passing Model
3. Data-Parallel Model
4. Object Oriented Model
5. Functional and Logic Model

1. Shared-Variable Model

- In all programming system, processors are **active resources** and memory & IO devices are **passive resources**. Program is a collection of processes. Parallelism depends on how IPC(Interprocess Communication) is implemented. Process address space is shared.
- To ensure orderly IPC, a mutual exclusion property requires that shared object must be shared by only 1 process at a time.

Shared Variable communication

- Used in multiprocessor programming
- Shared variable IPC demands use of shared memory and mutual exclusion among multiple processes accessing the same set of variables.

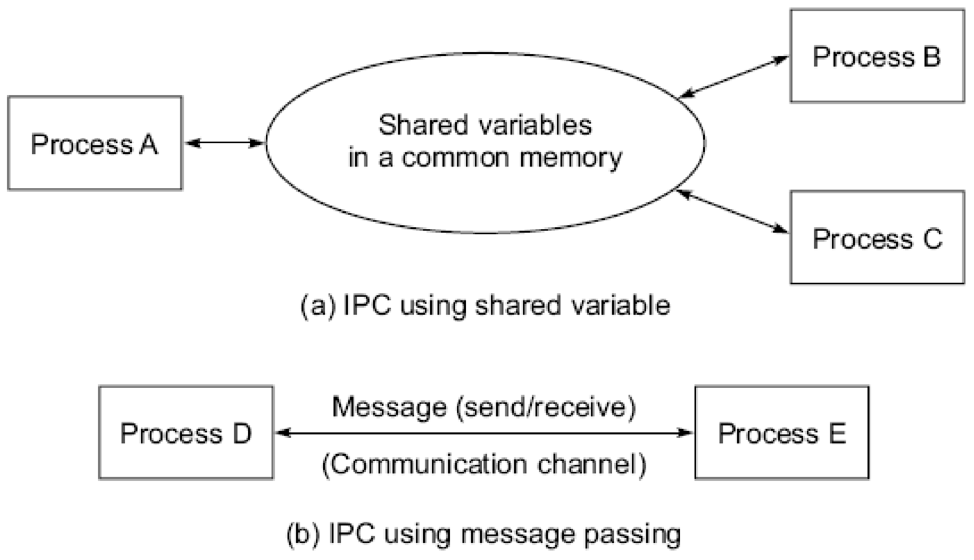

Process A
Shared variables in a common memory
Process B
Process C
(a) IPC using shared variable
Process D
Message (send/receive)
(Communication channel)
Process E
(b) IPC using message passing

Fig. 10.1 Two basic mechanisms for interprocess communication (IPC).

Critical Section

- Critical Section(CS) is a code segment accessing shared variable, which must be executed by only one process at a time and which once started must be completed without interruption.
- It should satisfy following requirements-:

✓ **Mutual Exclusion:** At most one process executing CS at a time.

✓ **No deadlock in waiting**: No circular wait by 2 or more process.

✓ **No preemption:** No interrupt until completion.

✓ **Eventual Entry**: Once entered CS,must be out after completion.

Protected Access

- Granularity of CS affects the performance.
- If CS is too large,it may limit parallism due to excessive waiting by process.
- When CS is too small,it may add unnecessary code complexity/Software overhead.

4 operational Modes

- Multiprogramming
- Multiprocessing
- Multitasking

Multithreading

2. Message Passing Model

Two processes D and E residing at different processor nodes may communicate wit each other by passing messages through a direct network. The messages may be instructions, data,synchronization or interrupt signals etc. Multicomputers are considered loosely coupled multiprocessors.

Synchronous Message Passing

- No shared Memory
- No mutual Exclusion
- Synchronization of sender and reciever process just like telephone call.
- No buffer used.
- If one process is ready to cummunicate and other is not,the one that is ready must be blocked.

Asynchronous Message Passing

- Does not require that message sending and receiving be synchronised in time and space.
- Arbitrary communication delay may be experienced because sender may not know if and when the message has been received until acknowledgement is received from receiver.
- This scheme is like a postal service using mailbox with no synchronization between senders and receivers.

3. Data Parallel Model

- Used in SIMD computers. Parallelism handled by hardware synchronization and flow control.
- Fortran 90 ->data parallel lang.
- Require predistrubuted data sets.

Data Parallelism

- This technique used in array processors(SIMD)
- Issue->match problem size with machine size.

Array Language Extensions

- Various data parallel language used
- Represented by high level data types
- CFD for Illiac 4,DAP fortran for Distributed array processor,C* for Connection machine
- Target to make the number of PE’s of problem size.

4. Object Oriented Model

- Objects dynamically created and manipulated.
- Processing is performed by sending and receiving messages among objects.

Concurrent OOP

- Need of OOP because of abstraction and reusability concept.

- Objects are program entities which encapsulate data and operations in single unit.
- Concurrent manipulation of objects in OOP.

Actor Model

- This is a framework for Concurrent OOP.
- Actors -> independent component
- Communicate via asynchronous message passing.
- 3 primitives -> create, send-to and become.

Parallelism in COOP

3 common patterns for parallelism-:

1) Pipeline concurrency
2) Divide and conquer
3) Cooperative Problem Solving

5. Functional and logic Model

- Functional Programming Language-> Lisp,Sisal and Strand 88.
  Logic Programming Language-> Concurrent Prolog and Parlog

Functional Programming Model

- Should not produce any side effects.
- No concept of storage,assignment and branching.
- Single assignment and data flow language functional in nature.

Logic Programming Models

- Used for knowledge processing from large database.
- Supports implicitly search strategy.
- And parallel execution and Or Parallel Reduction technique used.
- Used in artificial intelligence

10.2 Parallel Languages and Compilers

- ✓ Programming environment is collection of s/w tools and system support.
- ✓ Parallel Software Programming environment needed.
- ✓ Users still forced to focus on hardware details rather than parallelism using high level abstraction.

10.2.1 Language Features for Parallelism

Language features for parallel programming for parallel programming into 6 categories:

1. Optimization Features
2. Availability Features
3. Synchronization/communication Features
4. Control Of Parallelism
5. Data Parallelism Features
6. Process Management Features

1. Optimization Features

- Conversion of sequential Program to Parallel Program.
- The purpose is to match s/w parallelism with hardware parallelism.
- Software in Practice-:

1) Automated Parallelizer

Express C automated parallelizer and Allaint FX Fortran compiler.

2) Semiautomated Parallizer

Needs compiler directives or programmers interaction.

2. Availability Features

Enhance user friendliness, make language portable for large no of parallel computers and expand the applicability of software libraries.

1) Scalability

Language should be scalable to number of processors and independent of hardware topology.

2) Compatibility

Compatible with sequential language.

3) Portability

Language should be portable to shared memory multiprocessor, message passing or both.

3. Synchronization/Communication Features

- Shared Variable (locks) for IPC
- Remote Procedure Calls
- Data Flow languages

- Mailbox,Semaphores,Monitors

4. Control of Parallelism

- Coarse,Medium and fine grain
- Explicit vs implicit parallelism
- Global Parallelism
- Loop Parallelism
- Task Parallelism
- Divide and Conquer Parallelism

5. Data Parallelism Features

How data is accessed and distributed in either SIMD and MIMD computers.

- Runtime automatic decomposition

Data automatically distributed with no user interaction.

- Mapping Specification

User specifies patterns and input data mapped to hardware.

- Virtual Processor Support

Compilers made statically and maps to physical processor.

- Direct Access to shared data

Shared data is directly accessed by operating system.

6. Process Management Features

Support efficient creation of parallel process,multithreading/multitasking,program partitioning and replication and dynamic load balancing at run time.

1) Dynamic Process Creation at Run Time.

2) Creation of lightweight processes.

3) Replication technique.

4) Partitioned Networks.

5) Automatic Load Balancing

10.2.2 Parallel Language Constructs

Special language constructs and data array expressions ar presented below for exploiting parallelism in programs.

Fortran 90 Array Notation

A multidimensional data array is represented by an array name indexed by a sequence of subscript triplets, one for each dimension. Triplets for different dimensions separated by commas.

Examples are:

$\mathbf{e_1 : e_2 : e_3}$

$\mathbf{e_1 : e_2}$

$\mathbf{e_1 : * : e_3}$

$\mathbf{e_1 : *}$

$\mathbf{e_1}$

*****

where each e1 is an arithmetic expression that must produce a scalar integer value. The first expression e1 is a lower bound, the second e2 an upper bound and the third e3 an increment (stride).

For example, **B(1:4:3, 6:8:2, 3)** represents four elements **B(1, 6, 3), B(4, 6, 3), B(1, 8, 3),** and **B(4, 8, 3),** of a three-dimensional array.

When the third expression in a triplet is missing, a unit stride is assumed. The * notation in the second expression indicates all elements in that dimension starting from e1 or the entire dimension if e1 is also omitted.

Parallel Flow Control

- The conventional Fortran Do loop declares that all scalar instructions within the **(Do, Enddo)** pair are executed sequentially, and so are the successive iterations.
- To declare parallel activities, we use the **(Doall, Endall)** pair.
- All iterations in the **Doall** loop are totally independent of each other. They can be executed in parallel if there are sufficient resources.
- When the successive loops depend on each other, we use the **(Doacross, EndAcross)** pair to declare parallelism with loop-carried dependences.

- The **(ForAll, EndAll)** and **(ParDo, ParEnd)** commands can be interpreted either as a **Doall** loop or as a **Doacross** loop.

```
Doacross I=2, N
     Do J=2, N
S1:       A9I, J) = (A(I, J-1)) + A(I, J+1)) / 2
     Enddo
Endacross
```

Another program construct is **(Cobegin, Coend)** pair. All computations specified within the block could be executed in parallel.

```
Cobegin
P1
P2
…..
Pn
Coend
```

Causes processes **P1, P2,… Pn** to start simultaneously and to proceed concurrently until they have all ended. The command **(Parbegin, Parend)** has equivalent meaning.

During the execution of a process, we can use a **Fork Q** command to spawn a new process Q:

```
Process P                Process Q
…..                      ………
Fork Q                   ………
……                       End
Join Q
```

The **Join Q** command recombines the two processes into one process.

10.2.3 Optimizing Compilers for Parallelism

- Role of compiler to remove burden of optimization and generation.

  3 Phases are-:

  1) Flow analysis
  2) Optimization
  3) Code Generation

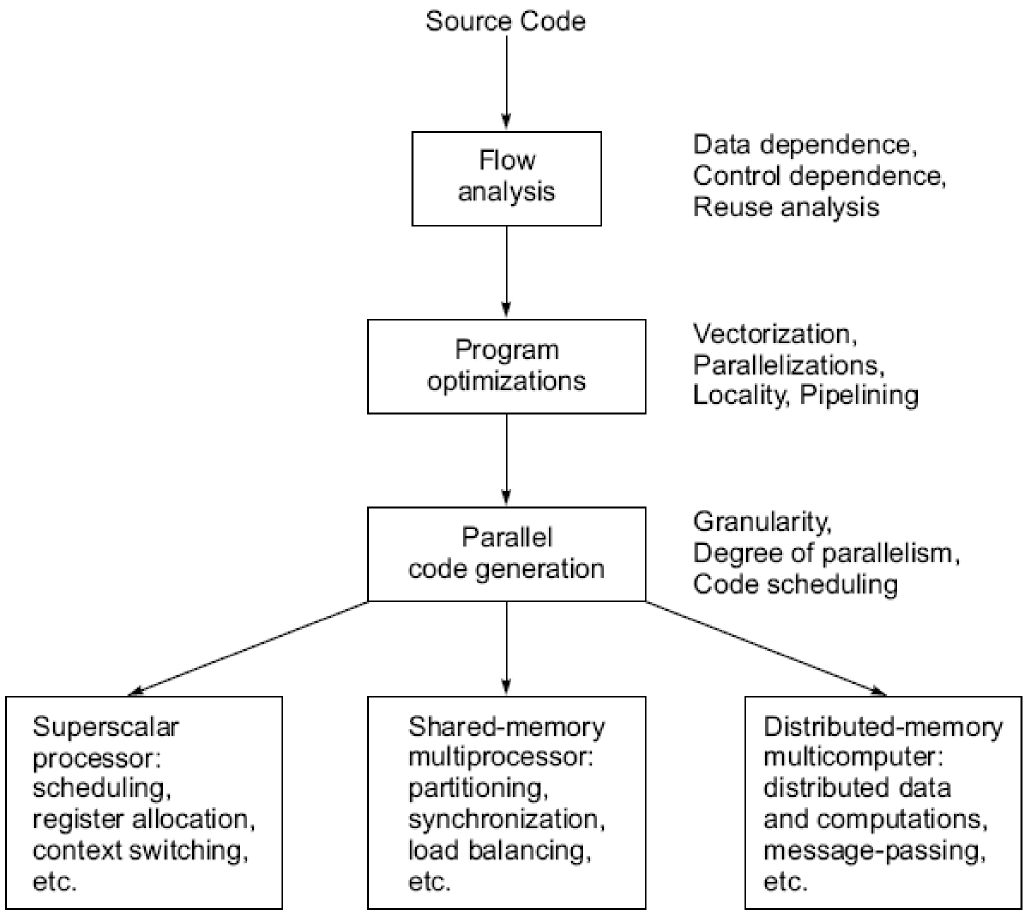

Source Code
Flow analysis
Data dependence, Control dependence, Reuse analysis
Program optimizations
Vectorization, Parallelizations, Locality, Pipelining
Parallel code generation
Granularity, Degree of parallelism, Code scheduling
Superscalar processor: scheduling, register allocation, context switching, etc.
Shared-memory multiprocessor: partitioning, synchronization, load balancing, etc.
Distributed-memory multicomputer: distributed data and computations, message-passing, etc.

Fig. 10.4 Compilation phases in parallel code generation

1) Flow analysis

- Reveals design flow patters to determine data and control dependencies.
- Flow analysis carried at various execution levels.

  1)Instruction level->VLSI or superscaler processors.

  2)Loop level->Simd and systolic computer

  3)Task level->Multiprocessor/Multicomputer

2) Optimization

- Transformation of user program to explore hardware capability.
- Explores better performance.
- Goal to maximise speed of code execution.
- To minimize code length.
- Local and global optimizations.

  Machine dependent Transformation

3) Parallel Code Generation

Compiler directive can be used to generate parallel code.

- 2 optimizing compilers-:

  1) Parafase and Parafase 2

  2) PFC and Parascope

Parafase and Parafase2

- Transforms sequential programs of fortran 77 into parallel programs.
- Parafase consists of 100 program that are encoded and passed.
- Pass list indentifies dependencies and converts it to concurrent program.
- Parafase2 for c and pascal in extension to fortran.

PFC AND Parascope

- Translates Fortran 77 to Fortran 90 code.
- PFC package extended to PFC + for parallel code generation on shared memory multiprocessor.
- PFC performs analysis as following steps below-:
- PFC performs analysis as following steps below-:

  1) Inter-procedure Flow analysis

  2) Transformation

  3) Dependence analysis

  4) Vector Code Generation

10.3 Dependence Analysis of Data Arrays

(Refer Text book)

Chapter-11 Parallel Program Development and Environments

11.2 Synchronization and Multiprocessing Modes

11.2.1 Principles of Synchronization

- The performance and correctness of a parallel program execution rely heavily on efficient synchronization among concurrent computations in multiple processors.
- The source of synchronization problem is the sharing of writable objects (data structures) among processes. Once a writable object permanently becomes read-only, the synchronization problem vanishes at that point.
- Synchronization consists of implementing the order of operations in an algorithm by observing the dependences for writable data.
- Lowe-level synchronization primitives are often implemented directly in hardware. Resources such as the CPU, bus or network and memory units may also be involved in synchronization of parallel computations.

  The following methods are used for implementing efficient synchronization schemes.

    - Atomic Operations
    - Wait Protocols
    - Fairness policies'
    - Access order
    - Sole access protocols

11.2.2 Multiprocessor Execution Modes

Multiprocessor supercomputers are built for vector processing as ewll as for parallel processing across multiple processors.

Multiprocessing modes include parallel execution from the fine-grain process level to the medium-grain task level and to the coarse-grain program level.

Multiprocessing Requirements

- Fast context switching among multiple processes resident in processors
- Multiple register sets to facilitate context switching
- Fast memory access with conflict-free memory allocation

- Effective synchronization mechanism among multiple processors
- Software tools for achieving parallel processing and performance monitoring
- System and application software for interactive users.

Multitasking Environments

Multitasking exploits parallelism at several levels:

- Functional units are pipelined or chained together
- Multiiple functional units are closed concurrently
- I/O and CPU activities are overlapped
- Multiple CPUs cooperate on a single program to achieve minimal execution time

11.2.3 Multitasking on Cray Multiprocessors

Three levels of multitasking are:

1. Macrotasking
2. Microtasking
3. Autotasking

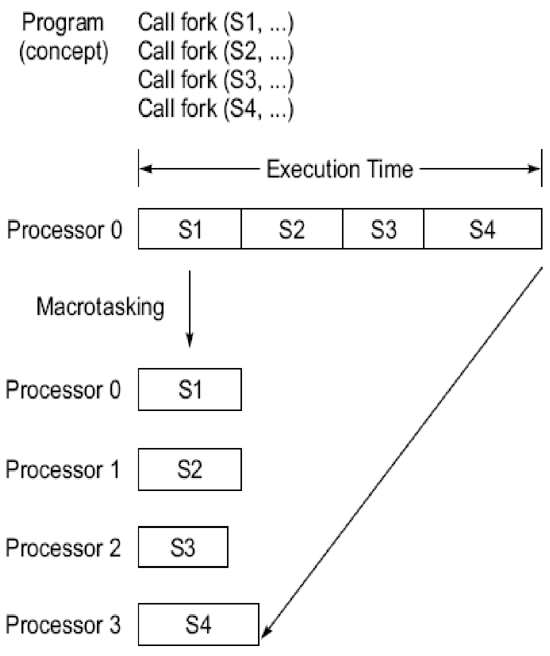
Macrotasking: When multitasking is conducted at the level of subroutine calls, it is called macrotasking with medium to coarse grains. The concept of macrotasking is shown in Fig 11.4a.

A main program forks a subroutine S1 and then forks out three additional subroutines S2, S3 and S4.

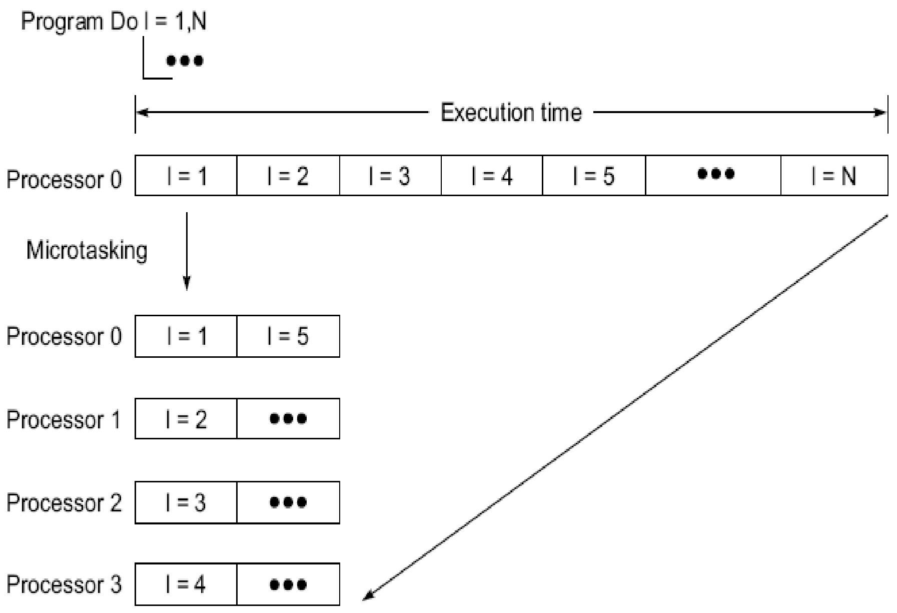
Microtasking: This corresponds to multitasking at the loop level with finer granularity. Compiler directives are often used to declare parallel execution of independent or dependent iterations of a looping program construct.

Fig 11.4b illustrates the spread of every four instructions of a Do loop to four processors simultaneously through microtasking.

Autotasking: It automatically divides a program into discrete tasks for parallel execution on a multiprocessor.



(a) Macrotasking across subroutine calls



(b) Microtasking across loop iterations

Fig. 11.4 Multitasking at two different processing levels

Chapter-12 Instruction Level Parallelism

12.1 Computer Architecture

(a) Computer Architecture is defined as the arrangement by which the various system building blocks—processors, functional units, main memory, cache, data paths, and so on—are interconnected and inter-operated to achieve desired *system performance.*

(b) Processors make up the most important part of a computer system. Therefore, in addition to (a), **processor design** also constitutes a central and very important element of computer architecture. Various functional elements of a processor must be designed, interconnected and inter-operated to achieve desired ***processor performance***.

- *System performance* is the key benchmark in the study of computer architecture. A computer system must solve the real world problem, or support the real world application, for which the user is installing it.
- A basic rule of system design is that *there should be no performance bottlenecks in the system*.
- Typically, a performance bottleneck arises when one part of the system.
- In a computer system, the key subsystems are processors, memories, I/O interfaces, and the data paths connecting them. Within the processors, we have subsystems such as functional units, registers, cache memories, and internal data buses.
- Within the computer system as a whole—or within a single processor—designers do not wish to create bottlenecks to system performance.

Example 12.1 Performance bottleneck in a system

In Fig. 12.1 we see the schematic diagram of a simple computer system consisting of four processors, a large shared main memory, and a processor-memory bus.

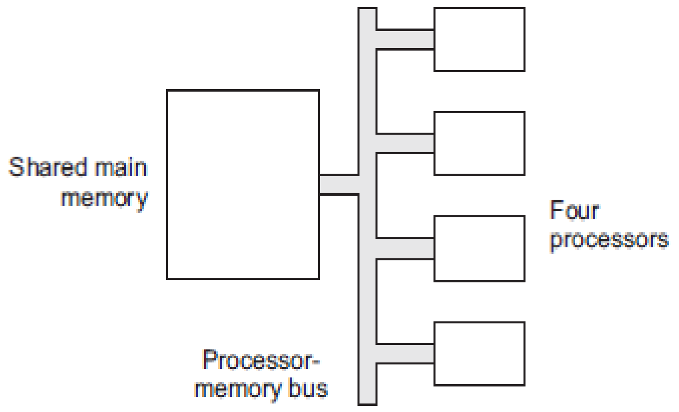


Fig. 12.1 A simple shared memory multiprocessor system

For the three subsystems, we assume the following performance figures:

(i) Each of the four processors can perform double precision floating point operations at the rate of 500 million per second, i.e. 500 MFLOPs.

(ii) The shared main memory can read/write data at the aggregate rate of 1000 million 32-bit words per second.

(iii) The processor-memory bus has the capability of transferring 500 million 32-bit words per second to/from main memory.

12.4 Model of a Typical Processor

- A processor with *load-store* instruction set architecture and a set of programmable registers as seen by the assembly language programmer or the code generator of a compiler.
- Whether these registers are bifurcated into separate sets of integer and floating point registers is not important for us at present, nor is the exact number of these registers.
- To support parallel access to instructions and data at the level of the fastest cache, we assume that L1 cache is divided into instruction cache and data cache, and that this split L1 cache supports single cycle access for instructions as well as data.
- Some processors may have an *instruction buffer* in place of L1 instruction cache; for the purposes of this section, however, the difference between them is not important.
- The first three pipeline stages on our prototype processor are ***fetch***, ***decode*** and ***issue***.
- Following these are the various functional units of the processor, which include integer unit(s), floating point unit(s), load/store unit(s), and other units as may be needed for a specific design.

- Let us assume that our superscalar processor is designed for *k* instruction issues in every processor clock cycle.
- Clearly then the *fetch*, *decode* and *issue* pipeline stages, as well as the other elements of the processor, must all be designed to process *k* instructions in every clock cycle.

- On multiple issue pipelines, *issue* stage is usually separated from *decode* stage. One reason for thus increasing a pipeline stage is that it allows the processor to be driven by a faster clock.
- *Decode* stage must be seen as preparation for instruction *issue* which—by definition—can occur only if the relevant functional unit in the processor is in a state in which it can accept one more operation for execution.
- As a result of the *issue*, the operation is handed over to the functional unit for execution.

- The process of issuing instructions to functional units also involves instruction scheduling. For example, if instruction I_j cannot be issued because the required functional unit is not free, then it may still be possible to issue the next instruction I_{j+1}—provided that no dependence between the two prohibits issuing instruction I_{j+1}.
- When instruction scheduling is specified by the compiler in the machine code it generates, we refer to it as *static scheduling*.
- In theory, static scheduling should free up the processor hardware from the complexities of instruction scheduling; in practice, though, things do not quite turn out that way.
- If the processor control logic schedules instruction *on the fly*—taking into account inter-instruction dependences as well as the state of the functional units—we refer to it as *dynamic scheduling*.
- Much of the rest of this chapter is devoted to various aspects and techniques of dynamic scheduling.
- The basic aim in both types of scheduling—static as well as dynamic—is to maximize the instruction level parallelism which is exploited in the executing sequence of instructions.
- The process of issuing instructions to functional units also involves instruction scheduling.
- A branch prediction unit has also been shown in Fig. 12.4 and Fig. 12.5 to implement some form of a branch prediction algorithm

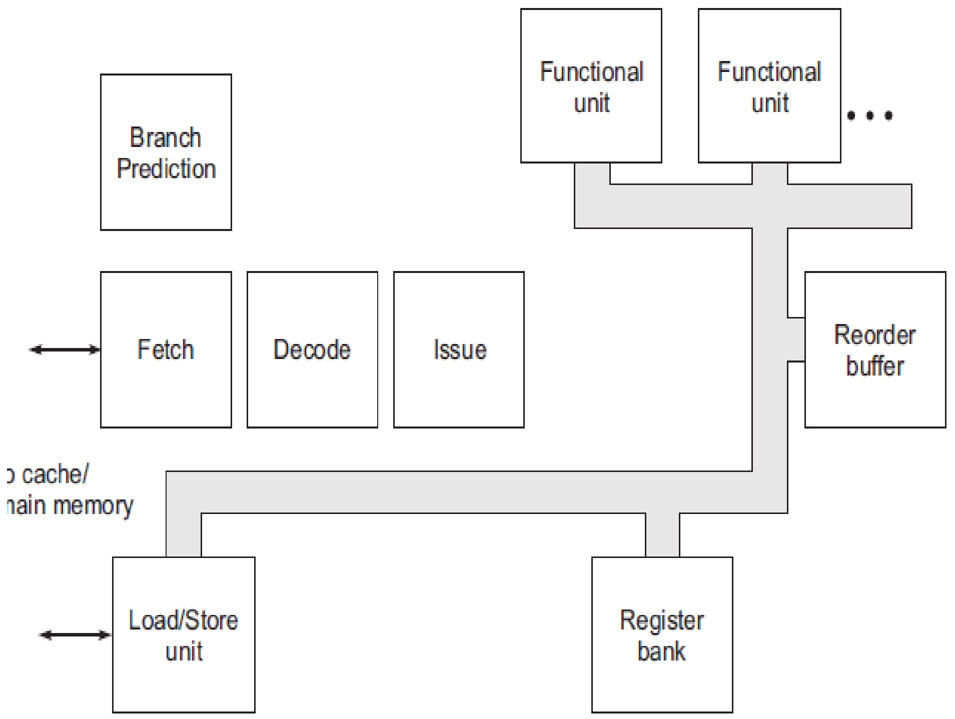


Fig. 12.4 Processor design with reorder buffer

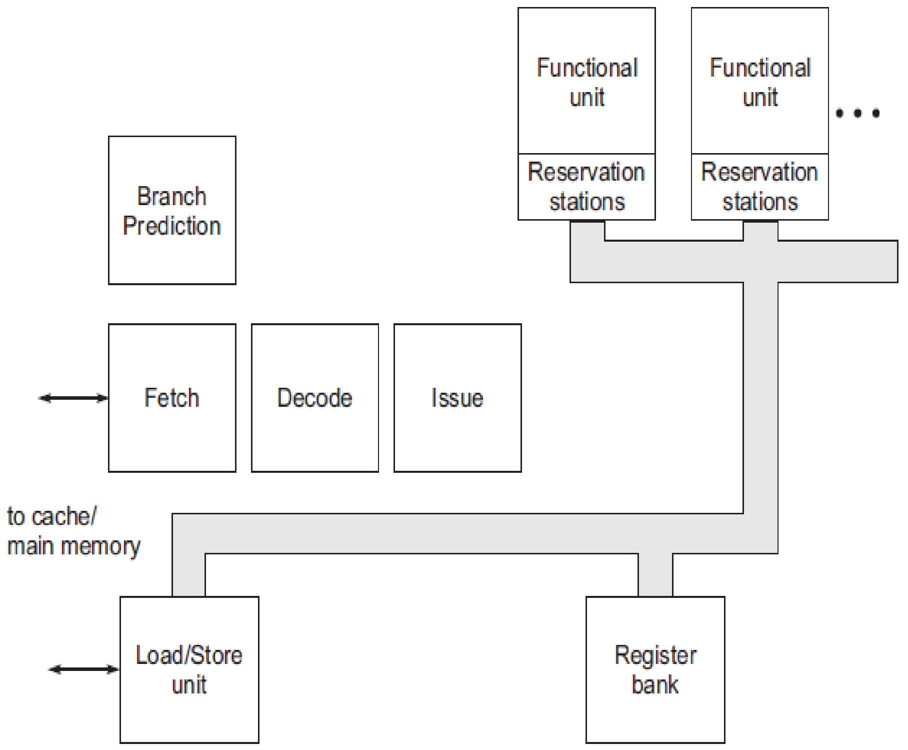


Fig. 12.5 Processor design with reservation stations on functional units

Figure 12.5 shows a processor design in which functional units are provided with reservation stations. Such designs usually also make use of operand forwarding over a common data bus (CDB), with tags to identify the source of data on the bus. Such a design also implies register renaming, which resolves RAW and WAW dependences.

12.5 Compiler-detected Instruction Level Parallelism

One relatively simple technique which the compiler can employ is known as loop unrolling, by which independent instructions from multiple successive iterations of a loop can be made to execute in parallel. Unrolling means that the body of the loop is repeated n times for n successive values of the control variable—so that one iteration of the transformed loop performs the work of n iterations of the original loop.

Loop Unrolling

Independent instructions from multiple successive iterations of a loop can be made to execute in parallel. *Unrolling* means that the body of the loop is repeated *n* times for *n* successive values of the control variable—so that one iteration of the transformed loop performs the work of *n* iterations of the original loop.

Consider the following body of a loop in a user program, where all the variables except the loop control variable i are assumed to be floating point:

```
for i = 0 to 58 do
        c[i] = a[i]*b[i] – p*d[i];
```

Now suppose that machine code is generated by the compiler as though the original program had been written as:

```
for j = 0 to 52 step 4 do
{
        c[j] = a[j]*b[j] – p*d[j];
        c[j + 1] = a[j + 1]*b[j + 1] – p*d[j + 1];
        c[j + 2] = a[j + 2]*b[j + 2] – p*d[j + 2];
        c[j + 3] = a[j + 3]*b[j + 3] – p*d[j + 3];
}
c[56] = a[56]*b[56] – p*d[56];
c[57] = a[57]*b[57] – p*d[57];
```

c[58] = a[58]*b[58] – p*d[58];

Note carefully the values of loop variable j in the transformed loop. the two program fragments are equivalent, in the sense that they perform the same computation.

Of course the compiler does not transform one source program into another—it simply produces machine code corresponding to the second version, with the *unrolled* loop.

12.6 Operand Forwarding

It helps in reducing the impact of true data dependences in the instruction stream. Consider the following simple sequence of two instructions in a running program:

ADD R1, R2, R3

SHIFTR #4, R3, R4

The result of the ADD instruction is stored in destination register R3, and then shifted right by four bits in the second instruction, with the shifted value being placed in R4.

Thus, there is a simple RAW *dependence* between the two instructions—the output of the first is required as input operand of the second represented in the form of graph as below:

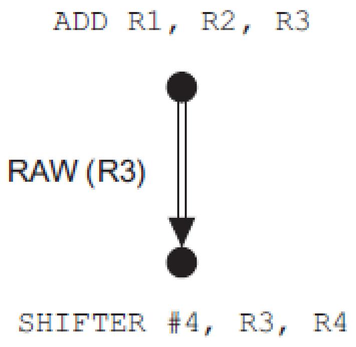


Fig. 12.6 RAW dependence between two instructions

- In a pipelined processor, ideally the second instruction should be executed one stage—and therefore one clock cycle—behind the first.
- However, the difficulty here is that it takes one clock cycle to transfer ALU output to destination register R3, and then another clock cycle to transfer the contents of register R3 to ALU input for the right shift. Thus a total of two clock cycles are needed to bring the result of the first instruction where it is needed for the second instruction.

- Therefore, as things stand, the second instruction above cannot be executed just one clock cycle behind the first.
- This sequence of data transfers has been illustrated in Fig. 12.7 (a). In clock cycle Tk, ALU output is transferred to R3 over an internal data path. In the next clock cycle Tk + 1, the content of R3 is transferred to ALU input for the right shift.
- When carried out in this order, clearly the two data transfer operations take two clock cycles.

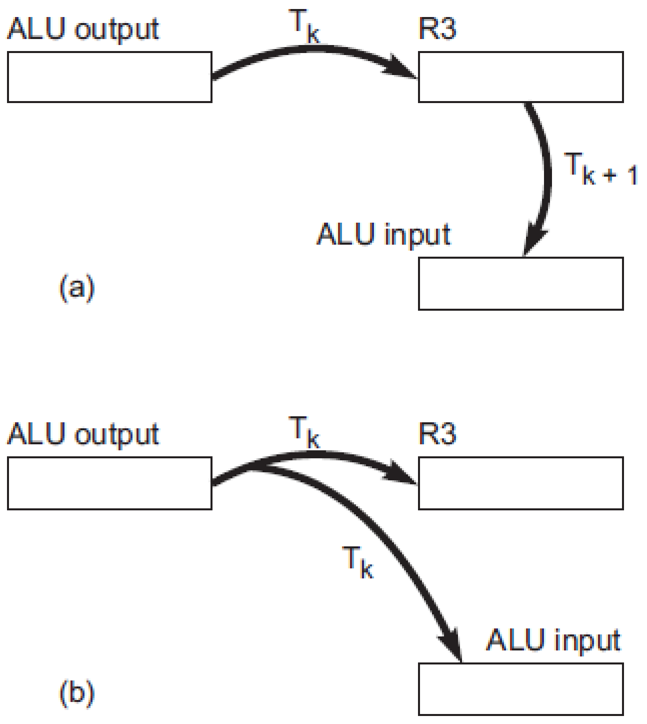


Fig. 12.7 Two data transfers (a) in sequence and (b) in parallel.

- But note that the required two transfers of data can be achieved in only one clock cycle if ALU output is sent to both R3 and ALU input in the same clock cycle—as illustrated in Fig. 12.7 (b).
- In general, if X is to be copied to Y, and in the next clock cycle Y is to be copied to Z, then we can just as well copy X to both Y and Z in one clock cycle.

12.7 Reorder Buffer

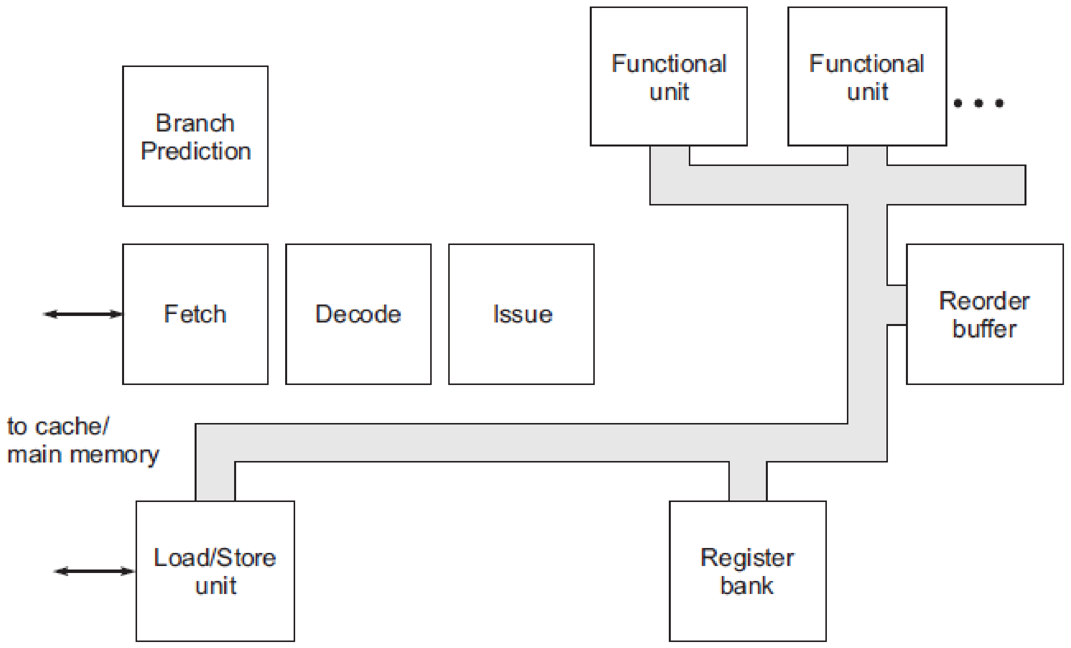
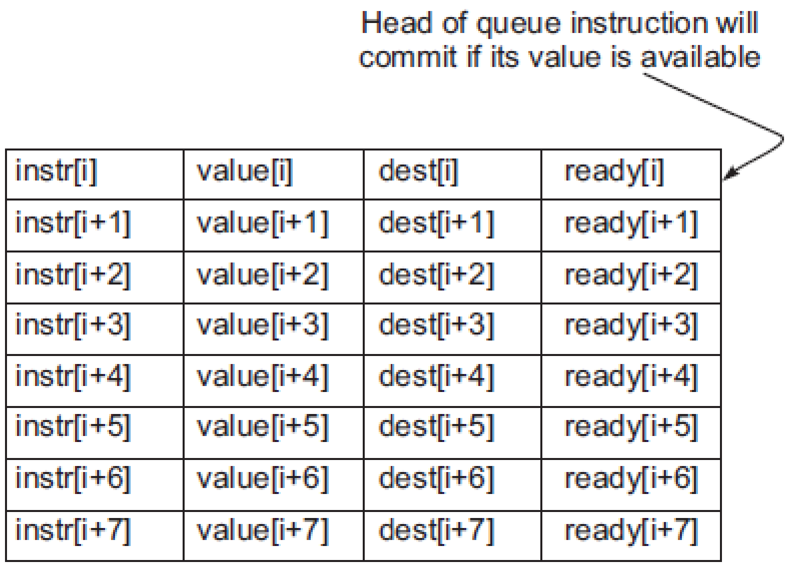


Fig. 12.4 Processor design with reorder buffer

- Since instructions execute in parallel on multiple functional units, the reorder buffer serves the function of bringing completed instructions back into an order which is consistent with program order.
- Note that instructions may *complete* in an order which is not related to program order, but must be *committed* in program order.
- At any time, *program state* and *processor state* are defined in terms of instructions which have been committed—i.e. their results are reflected in appropriate registers and/or memory locations.
- Entries in the reorder buffer are completed instructions, which are queued in program order.
- However, since instructions do not necessarily complete in program order, we also need a flag with each reorder buffer entry to indicate whether the instruction in that position has completed.


Head of queue instruction will commit if its value is available

instr[i]	value[i]	dest[i]	ready[i]
instr[i+1]	value[i+1]	dest[i+1]	ready[i+1]
instr[i+2]	value[i+2]	dest[i+2]	ready[i+2]
instr[i+3]	value[i+3]	dest[i+3]	ready[i+3]
instr[i+4]	value[i+4]	dest[i+4]	ready[i+4]
instr[i+5]	value[i+5]	dest[i+5]	ready[i+5]
instr[i+6]	value[i+6]	dest[i+6]	ready[i+6]
instr[i+7]	value[i+7]	dest[i+7]	ready[i+7]

Fig. 12.9 Entries in a reorder buffer of size eight

Figure 12.9 shows a reorder buffer of size eight. Four fields are shown with each entry in the reorder buffer—instruction identifier, value computed, program-specified destination of the value computed, and a flag indicating whether the instruction has completed (i.e. the computed value is available).

We now take a brief look at how the use of reorder buffer addresses the various types of dependences in the program.

1. **Data Dependences:** A RAW dependence—i.e. true data dependence—will hold up the execution of the dependent instruction if the result value required as its input operand is not available. As suggested above, operand forwarding can be added to this scheme to speed up the supply of the needed input operand as soon as its value has been computed.

WAR and WAW dependences—i.e. anti-dependence and output dependence, respectively—also hold up the execution of the dependent instruction and create a possible pipeline stall. We shall see below that the technique of register renaming is needed to avoid the adverse impact of these two types of dependences.

2**. Control Dependences:**

Suppose the instruction(s) in the reorder buffer belong to a branch in the program which should not have been taken—i.e. there has been a mis-predicted branch. Clearly then the reorder buffer should be flushed along with other elements of the pipeline.

Therefore the performance impact of control dependences in the running program is determined by the accuracy of branch prediction technique employed.

The reorder buffer plays no direct role in the handling of control dependences.

3. **Resource Dependences:** If an instruction needs a functional unit to execute, but the unit is not free, then the instruction must wait for the unit to become free—clearly no technique in the world can change that.

In such cases, the processor designer can aim to achieve at least this: if a subsequent instruction needs to use another functional unit which is free, then the subsequent instruction can be executed out of order.

12.8 Register Renaming

- Traditional compilers and assembly language programmers work with a fairly small number of programmable registers.
- Therefore the only way to make a larger number of registers available to instructions under execution within the processor is to make the additional registers *invisible* to machine language instructions.
- Instructions *under execution* would use these additional registers, even if instructions making up the machine language program stored in memory cannot refer to them.

For example, let us say that the instruction:

FADD R1, R2, R5

is followed by the instruction:

FSUB R3, R4, R5

Both these instructions are writing to register R5, creating thereby a WAW dependence—i.e. output dependence—on register R5.

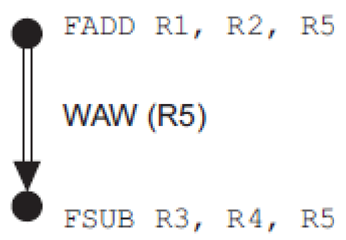


Fig. 12.10 WAW dependence

- Let FSUB write its output value to a register other than R5, and let us call that other register X. Then the instructions which use the value generated by FSUB will refer to X, while the instructions which use the value generated by FADD will continue to refer to R5.
- Now, since FADD and FSUB are writing to two different registers, the output dependence or WAW between them has been removed.

- When FSUB *commits*, then the value in R5 should be updated by the value in X—i.e. the value computed by FSUB.
- Then the physical register X, which is not a program visible register, can be freed up for use in another such situation.
- Note that here we have *mapped*—or *renamed*—R5 to X, for the purpose of storing the result of FSUB, and thereby removed the WAW dependence from the instruction stream. A pipeline stall will now not be created due to the WAW dependence.

12.9 Tomasulo's Algorithm

- *Register renaming* was also an implicit part of the original algorithm.
- For register renaming, we need a set of program invisible registers to which programmable registers are re-mapped.
- Tomasulo's algorithm requires these program invisible registers to be provided with reservation stations of functional units.
- Let us assume that the functional units are internally pipelined, and can complete one operation in every clock cycle.
- Therefore each functional unit can initiate one operation in every clock cycle—provided of course that a reservation station of the unit is ready with the required input operand value or values.
- Note that the exact depth of this functional unit pipeline does not concern us for the present.
- Figure 12.12 shows such a functional unit connected to the common data bus, with three reservation stations provided on it.

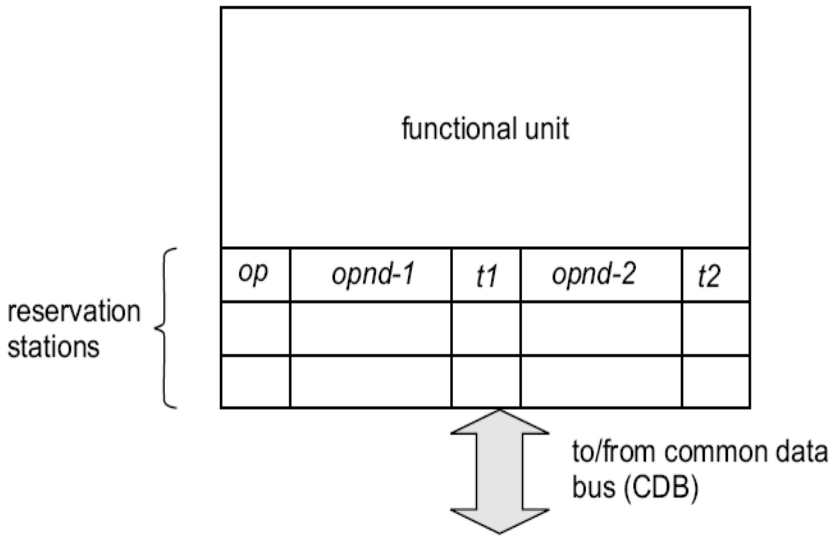


Fig. 12.12 Reservation stations provided with a functional unit

The various fields making up a typical reservation station are as follows:

op	operation to be carried out by the functional unit
opnd-1 & **opnd-2**	two operand values needed for the operation
t1 & t2	two source tags associated with the operands

- When the needed operand value or values are available in a reservation station, the functional unit can initiate the required operation in the next clock cycle.
- At the time of instruction issue, the reservation station is filled out with the operation code (*op*).
- If an operand value is available, for example in a programmable register, it is transferred to the corresponding source operand field in the reservation station.
- However, if the operand value is not available at the time of issue, the corresponding source tag (*t1* and/or *t2*) is copied into the reservation station.
- The source tag identifies the source of the required operand. As soon as the required operand value is available at its source—which would be typically the output of a functional unit—the data value is forwarded over the common data bus, along with the source tag.
- This value is copied into all the reservation station operand slots which have the matching tag.
- Thus operand forwarding is achieved here with the use of tags. All the destinations which require a data value receive it in the same clock cycle over the common data bus, by matching their stored operand tags with the source tag sent out over the bus.

Tomasulo's algorithm and RAW dependence

- Assume that instruction I1 is to write its result into R4, and that two subsequent instructions I2 and I3 are to read—i.e. make use of—that result value.
- Thus instructions I2 and I3 are truly data dependent (RAW dependent) on instruction I1. See Fig. 12.13.
- Assume that the value in R4 is not available when I2 and I3 are issued; the reason could be, for example, that one of the operands needed for I1 is itself not available.
- Thus we assume that I1 has not even started executing when I2 and I3 are issued.
- When I2 and I3 are issued, they are parked in the reservation stations of the appropriate functional units.
- Since the required result value from I1 is not available, these reservation station entries of I2 and I3 get source tag corresponding to the output of I1—i.e. output of the functional unit which is performing the operation of I1.
- When the result of I1 becomes available at its functional unit, it is sent over the common data bus along with the tag value of its source—i.e. output of functional unit.
- At this point, programmable register R4 as well as the reservation stations assigned to I2 and I3 have the matching source tag—since they are waiting for the same result value, which is being computed by I1.

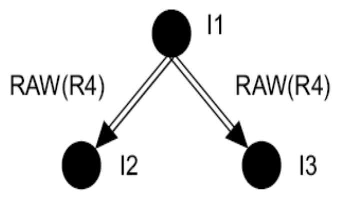


Fig. 12.13 Example of RAW dependences

- When the tag sent over the common data bus matches the tag in any destination, the data value on the bus is copied from the bus into the destination.
- The copy occurs at the same time into all the destinations which require that data value. Thus R4 as well as the two reservation stations holding I2 and I3 receive the required data value, which has been computed by I1, at the same time over the common data bus.

- Thus, through the use of source tags and the common data bus, in one clock cycle, three destination registers receive the value produced by I1—programmable register R4, and the operand registers in the reservation stations assigned to I2 and I3.
- Let us assume that, at this point, the second operands of I2 and I3 are already available within their corresponding reservation stations.
- Then the operations corresponding to I2 and I3 can begin in parallel as soon as the result of I1 becomes available—since we have assumed here that I2 and I3 execute on two separate functional units.

12.10 Branch Prediction

- About 15% to 20% of instructions in a typical program are branch and jump instructions, including procedure returns.
- Therefore—if hardware resources are to be fully utilized in a superscalar processor—the processor must start working on instructions beyond a branch, even before the branch instruction itself has completed. This is only possible through some form of branch prediction.
- What can be the logical basis for branch prediction? To understand this, we consider first the reasoning which is involved if one wishes to predict the result of a tossed coin.

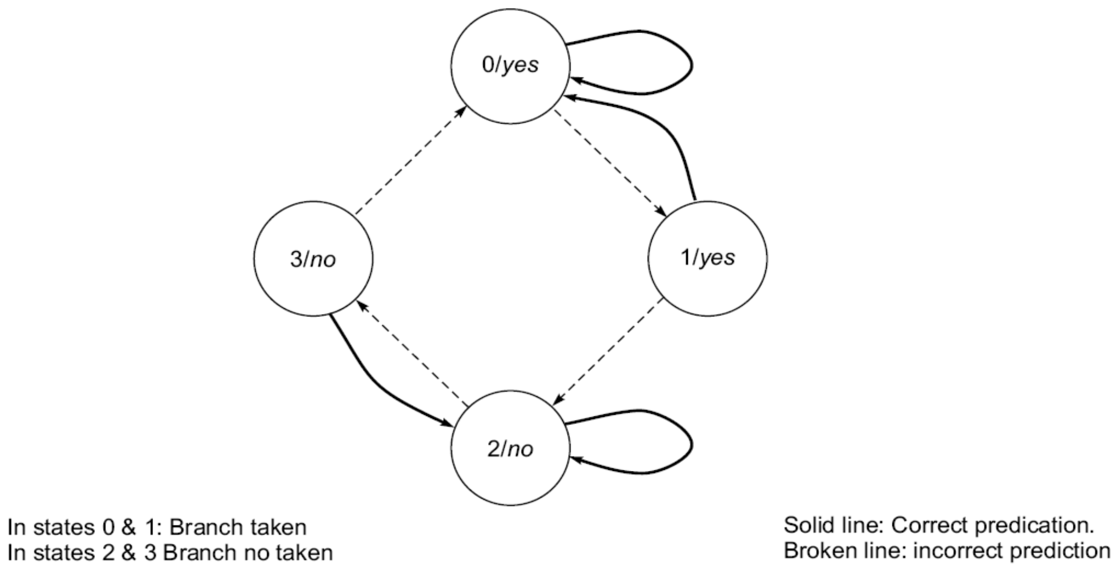

0/yes
1/yes
2/no
3/no
In states 0 & 1: Branch taken
In states 2 & 3 Branch no taken
Solid line: Correct predication.
Broken line: incorrect prediction

Fig. 12.15 State transition diagram of 2-bit branch predictor[8]

- A basic branch prediction technique uses a so-called *two-bit predictor*. A two-bit counter is maintained for every conditional branch instruction in the program. The two-bit counter has four possible states; these four states and the possible transitions between these states are shown in Fig. 12.15.

- When the counter state is 0 or 1, the respective branch is predicted as *taken*; when the counter state is 2 or 3, the branch is predicted as *not taken*.
- When the conditional branch instruction is executed and the actual branch outcome is known, the state of the respective two-bit counter is changed as shown in the figure using solid and broken line arrows.
- When two successive predictions come out wrong, the prediction is changed from *branch taken* to *branch not taken*, and *vice versa*.

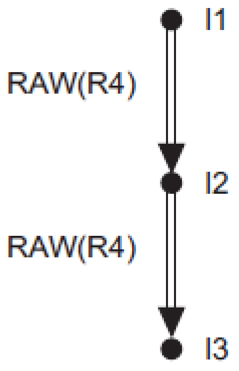


Fig. 12.14 Example of RAW & WAR dependences

- In Fig. 12.14, state transitions made on mis-predictions are shown using broken line arrows, while solid line arrows show state transitions made on predictions which come out right.
- This scheme uses a two-bit counter for every conditional branch, and there are many conditional branches in the program.
- Overall, therefore, this branch prediction logic needs a few kilobytes or more of fast memory.
- One possible organization for this branch prediction memory is in the form of an array which is indexed by low order bits of the instruction address.
- If twelve low order bits are used to define the array index, for example, then the number of entries in the array is 4096.
- To be effective, branch prediction should be carried out as early as possible in the instruction pipeline.
- As soon as a conditional branch instruction is decoded, branch prediction logic should predict whether the branch is taken.

- Accordingly, the next instruction address should be taken either as the branch target address (i.e. branch is taken), or the sequentially next address in the program (i.e. branch is not taken).

12.12 Thread Level Parallelism

- One way to reduce the burden of dependences is to combine—with hardware support within the processor—instructions from multiple independent threads of execution.
- Such hardware support for multi-threading would provide the processor with a pool of instructions, in various stages of execution, which have a relatively smaller number of dependences amongst them, since the threads are independent of one another.
- Let us consider once again the processor with instruction pipeline of depth eight, and with targeted superscalar performance of four instructions completed in every clock cycle.
- Now suppose that these instructions come from four independent threads of execution. Then, on average, the number of instructions in the processor at any one time from one thread would be 4 X 8/4 = 8.
- With the threads being independent of one another, there is a smaller total number of data dependences amongst the instructions in the processor.
- Further, with control dependences also being separated into four threads, less aggressive branch prediction is needed.
- Another major benefit of such hardware-supported multi-threading is that pipeline stalls are very effectively utilized.
- If one thread runs into a pipeline stall—for access to main memory, say—then another thread makes use of the corresponding processor clock cycles, which would otherwise be wasted.
- Thus hardware support for multi-threading becomes an important latency hiding technique.
- To provide support for multi-threading, the processor must be designed to switch between threads—either on the occurrence of a pipeline stall, or in a round robin manner.
- As in the case of the operating system switching between running processes, in this case the hardware context of a thread within the processor must be preserved.

Depending on the specific strategy adopted for switching between threads, hardware support for **multi-threading** may be classified as one of the following:

1. Coarse-grain multi-threading: It refers to switching between threads only on the occurrence of a major pipeline stall—which may be caused by, say, access to main memory, with latencies of the order of a hundred processor clock cycles.

2. Fine-grain multi-threading: It refers to switching between threads on the occurrence of any pipeline stall, which may be caused by, say, L1 cache miss. But this term would also apply to designs in which processor clock cycles are regularly being shared amongst executing threads, even in the absence of a pipeline stall.

3. Simultaneous multi-threading: It refers to machine instructions from two (or more) threads being issued in parallel in each processor clock cycle. This would correspond to a multiple-issue processor where the multiple instructions issued in a clock cycle come from an equal number of independent execution threads.