Lecture 22-24: September 27 and 29, 2021
Computer Architecture and Organization-II
Biplab K Sikdar

ILP -I

Target is to exploit full parallelism among the instructions in a program.
Improvement can be done in terms of the following factors:

- *Instruction issue rate*

  Controls number of instructions issued to pipeline per clock cycle (Figure 1).

  Improves through multiple issue pipeline structure.

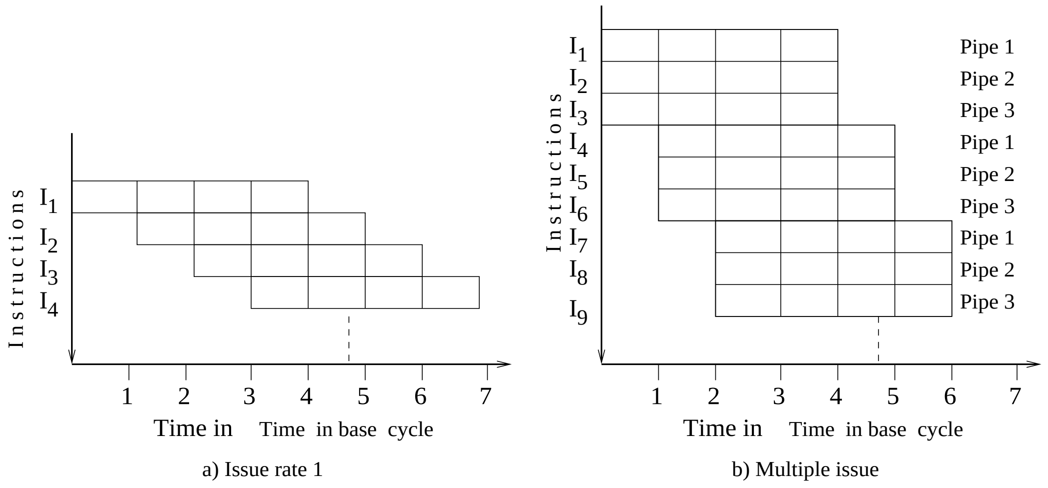


Figure 1: Instruction issue rate

- *Instruction issue latency*

  Time delays (in cycles) required between two successive issues of instruction in a pipeline (Figure 2).

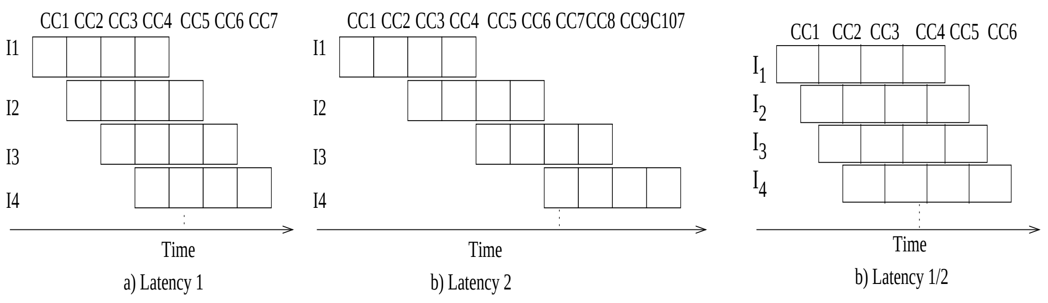


Figure 2: Instruction issue latency

- *Instruction level parallelism* (ILP)

  ILP is the measure of potential of instructions within a program that can be executed simultaneously.

- *Simple operation latency*

  A computer executes mostly simple operations such as

  Data movement, integer addition, load/store, branch etc.

  Execution of such instructions demands shorter latency.

- *Window of execution*:

  Set of instructions considered for execution at a time slice.

  Any instruction in the window is issued for parallel execution.

  More number of instructions in a window - better performance of the system.

0.1 Multiple Issue Processors

Goal is to set clock per instruction (CPI) <1.

There are two possibilities

a) *Static multiple-issue processors* (VLIW (*very large instruction word*))

   Decision to find set of simultaneous executable instructions - at compile time.

b) *Dynamic multiple-issue processors* (*superscalar processors*)

   Decisions on simultaneous executable instructions at run time (by hardware).

0.2 Superscalar Processors

The term superscalar, first coined in 1987.

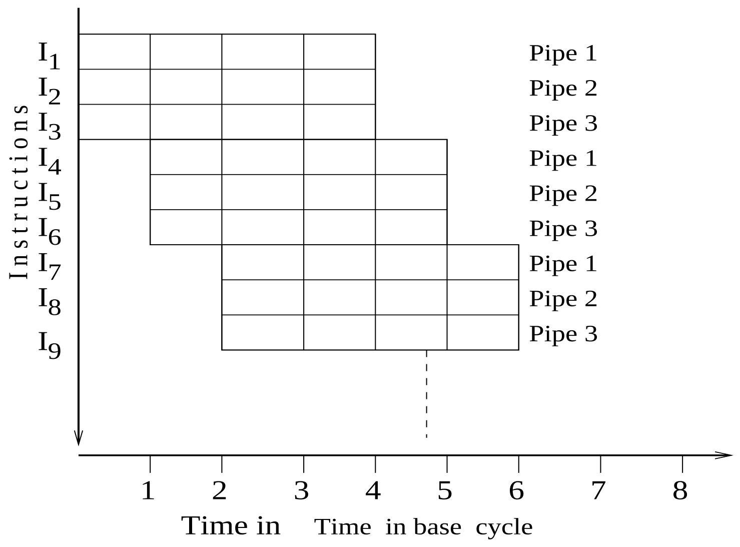


Figure 3: Superscalar execution

It has ability to execute instructions independently in different pipelines.

Multiple instruction pipelines are issued multiple instructions per cycle.

Number of instructions issued per cycle is the *degree* (m) of superscalar processor.

In Figure 3, issue rate m = 3 -that is, *degree* of superscalar execution is 3.

A general *basescalar pipelined processor* has issue rate m = 1 (Figure 4).

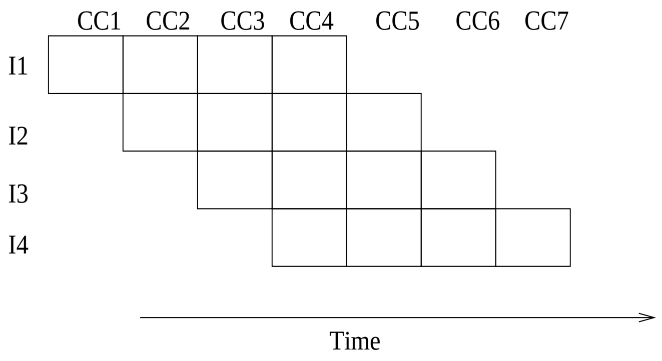
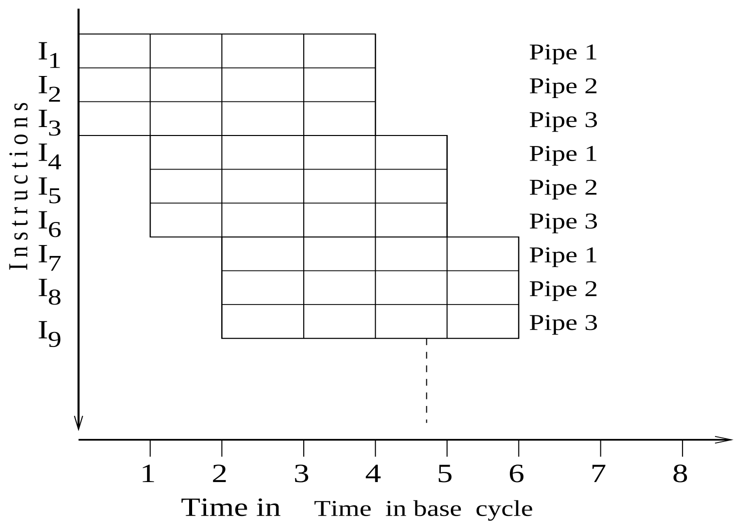


Figure 4: Basescalar execution

In superscalar, functional units (pipeline stages) are shared by multiple pipelines.



It reduces cost of design but increases possibility of structural hazards.

Sharing of functional units is shown in Figure 5.

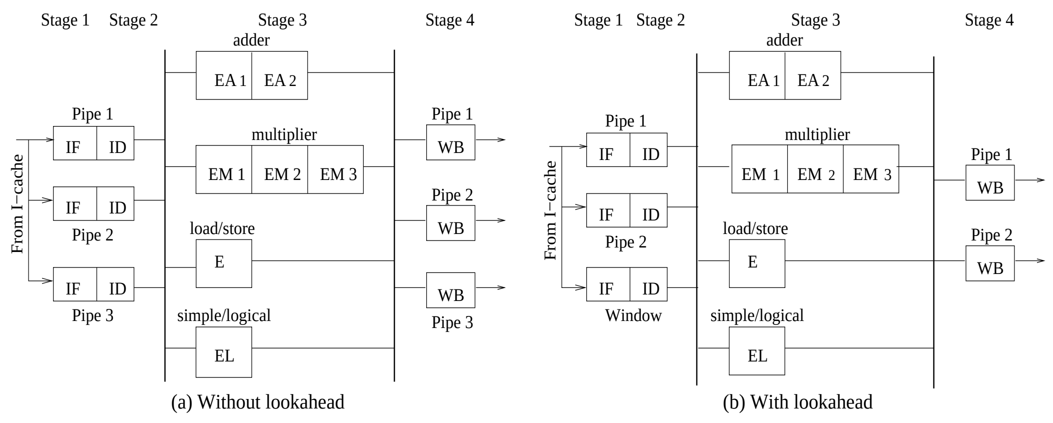


Figure 5: Three and two-issue superscalar processor architecture

A pipeline of Figure 5(a) has independent fetch (IF)/decode (ID)/store (WB) unit.

It implements 4-stage pipeline. It can issue three instructions per cycle.

Instruction streams for three pipelines are retrieved from I-cache.

These are to feed IF stages of Pipe 1, Pipe 2 and Pipe 3.

In EX, 4 functional units - multiplier, adder, logic unit and load/store unit are shared.

Multiplier unit itself has 3 pipeline stages EM1, EM2 and EM3.

Adder has two - EA1 and EA2.

Load/store unit has one E.

Logic unit has single stage EL.

Three WB are for 3 pipelines and can be dynamically used by the pipelines.

In Figure 5(b), window is used for instruction lookahead.

Degree m cannot always be improved by widening pipeline and h/w resources.

It also depends on fetch, decode and execute policies called *lookahead policy*.

Lookahead policy examines instructions beyond current one (I_i)

to find independent instructions (I_js) that can be executed independently with I_i.

Lookahead depends on instruction issue policies.

In-order issue

Dictates - instructions to be issued in an order as defined in program (Figure 6).

In-order completion

Defines - execution of instructions (also storing of results) is to be completed in same order as it appears in program. Follow Figure 6.

Out-of-order issue

An instruction I_{i+1} can be issued for execution prior to I_i. Follow Figure 6.

Out-of-order completion

Allows completion of instruction I_{i+1} prior to I_i. Follow Figure 6.

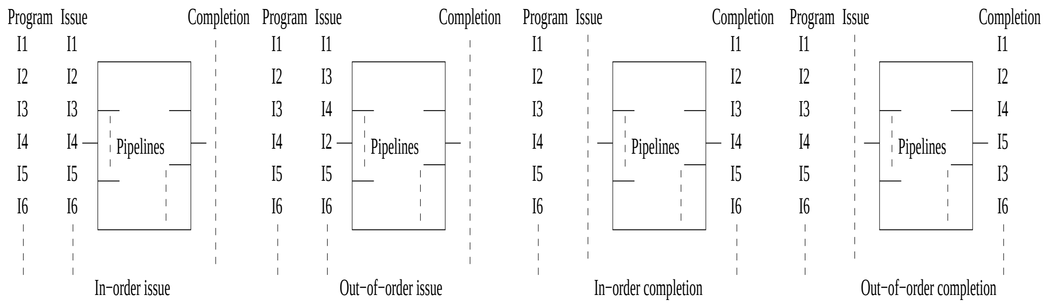


Figure 6: Instruction order

Superscalar processor category -

1. *In-order issue, in-order completion* superscalar computer,
2. *In-order issue, out-of-order completion* superscalar computer, and
3. *Out-of-order issue, out-of-order completion* superscalar computer.

Issue/completion order invites - data dependencies, control/ structural hazards.

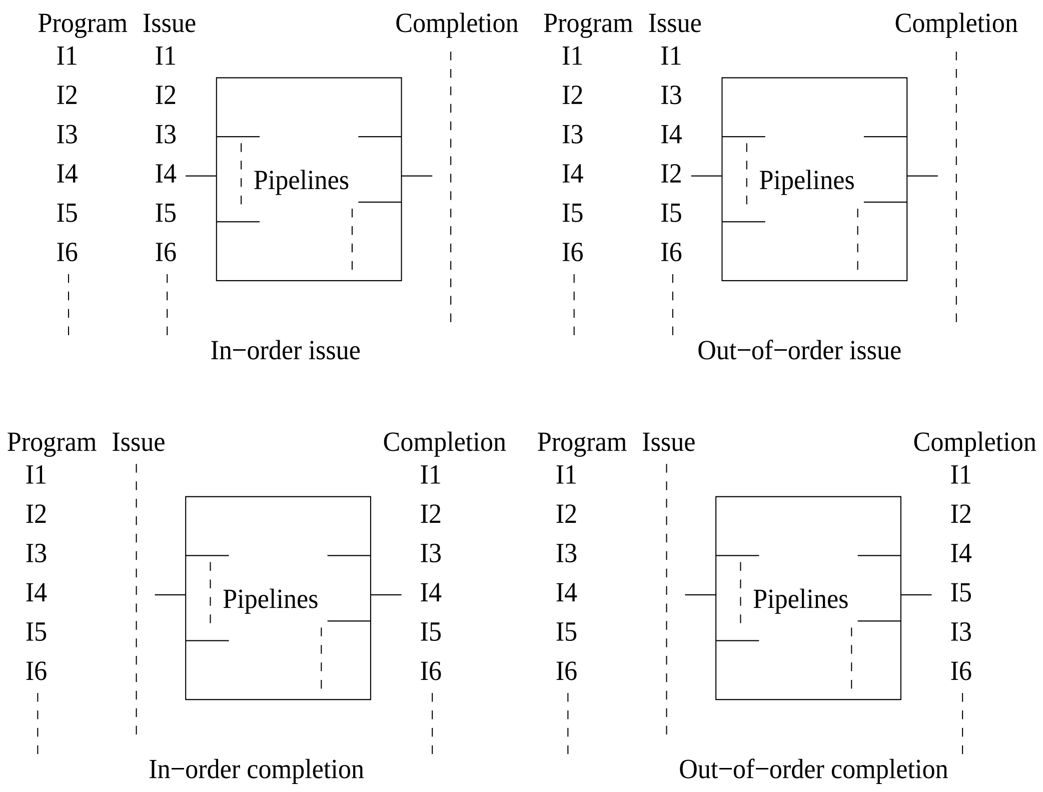


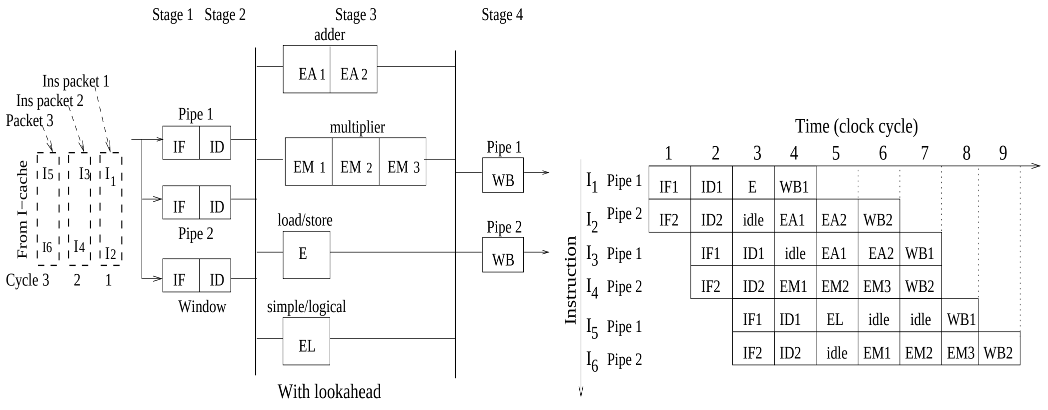
Figure 7: Instruction order

0.2.1 In-order issue in-order completion

Let consider the program segment:

$$
\begin{aligned}
I_1&: R_1 \leftarrow \text{Memory}(A)\\
I_2&: R_2 \leftarrow R_2 + R_1\\
I_3&: R_3 \leftarrow R_3 + R_4\\
I_4&: R_4 \leftarrow R_4 * R_5\\
I_5&: R_6 \leftarrow \neg R_6\\
I_6&: R_6 \leftarrow R_6 * R_7
\end{aligned}
$$

Assume 2-issue superscalar (Figure 8) processor with or without lookahead.



Instruction		1	2	3	4	5	6	7	8	9
I_1	Pipe 1	IF1	ID1	E	WB1					
I_2	Pipe 2	IF2	ID2	idle	EA1	EA2	WB2			
I_3	Pipe 1		IF1	ID1	idle	EA1	EA2	WB1		
I_4	Pipe 2		IF2	ID2	EM1	EM2	EM3	WB2		
I_5	Pipe 1			IF1	ID1	EL	idle	idle	WB1	
I_6	Pipe 2			IF2	ID2	idle	EM1	EM2	EM3	WB2

Figure 8: In-order issue in-order completion

Here is RAW between I_1 and I_2, WAR between I_3 and I_4, WAW between I_5 and I_6.

An idle cycle is introduced while executing I_2 to get operand R_1 loaded by I_1.

I_3 is delayed one (idle) cycle as the same adder is used by I_2 and I_3.

Finally, I_6 has to wait for result of I_5 before it can enter multiplier stages.

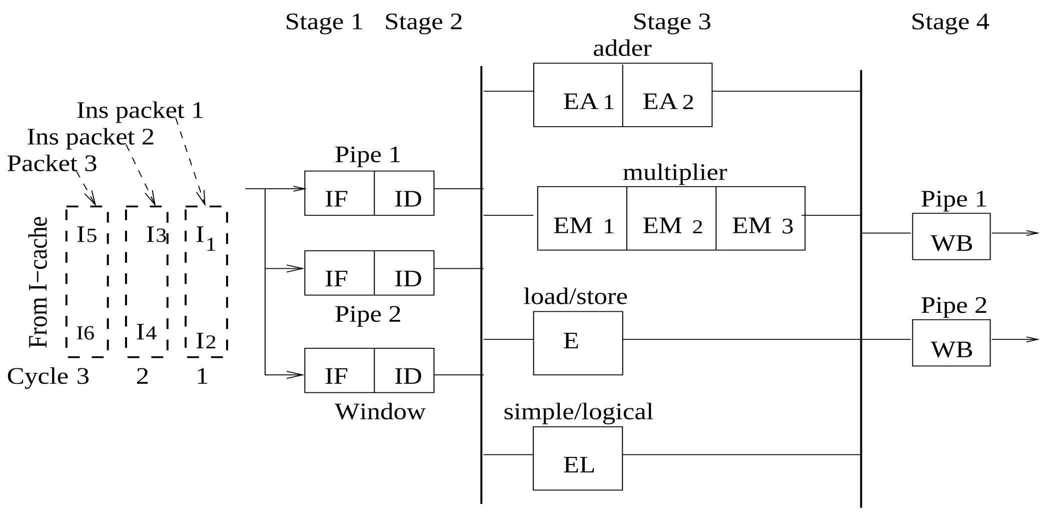
In-order to maintain in-order completion, I_5 is forced to wait for two (idle) cycles.

To complete execution, 9 cycles are needed out of which 5 are idle cycles.

In in-order issue, no new instruction can be issued when processor detects a conflict.

Pipeline is then stalled until conflict is resolved. No lookahead is allowed.

I_1: $R_1 \leftarrow \text{Memory}(A)$
I_2: $R_2 \leftarrow R_2 + R_1$
I_3: $R_3 \leftarrow R_3 + R_4$
I_4: $R_4 \leftarrow R_4 * R_5$
I_5: $R_6 \leftarrow \neg R_6$
I_6: $R_6 \leftarrow R_6 * R_7$



With lookahead

Time (clock cycle)

Instruction		1	2	3	4	5	6	7	8	9
I_1	Pipe 1	IF1	ID1	E	WB1					
I_2	Pipe 2	IF2	ID2	idle	EA1	EA2	WB2			
I_3	Pipe 1		IF1	ID1	idle	EA1	EA2	WB1		
I_4	Pipe 2		IF2	ID2	EM1	EM2	EM3	WB2		
I_5	Pipe 1			IF1	ID1	EL	idle	idle	WB1	
I_6	Pipe 2			IF2	ID2	idle	EM1	EM2	EM3	WB2

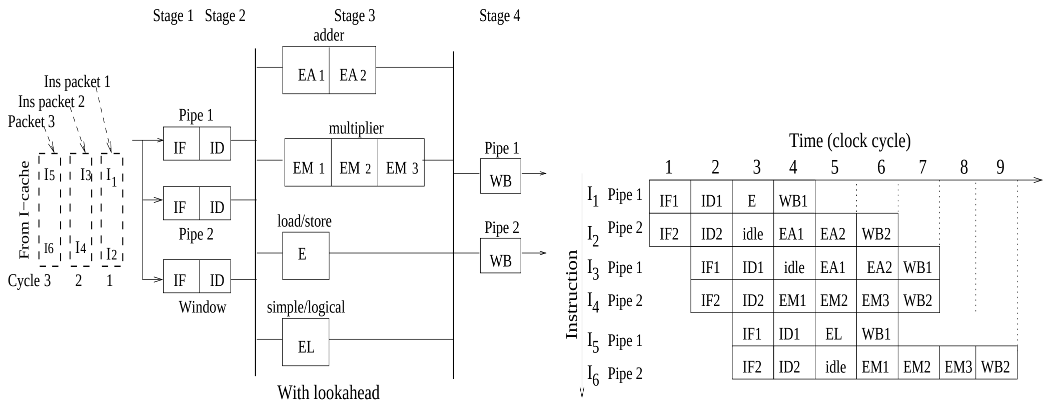
Figure 9: In-order issue in-order completion

0.2.2 In-order issue out-of-order completion

Follow

$$
\begin{aligned}
I_1&: R_1 \leftarrow \text{Memory(A)}\\
I_2&: R_2 \leftarrow R_2 + R_1\\
I_3&: R_3 \leftarrow R_3 + R_4\\
I_4&: R_4 \leftarrow R_4 * R_5\\
I_5&: R_6 \leftarrow \neg R_6\\
I_6&: R_6 \leftarrow R_6 * R_7
\end{aligned}
$$

Assume 2-issue superscalar processor with or without lookahead window.



		1	2	3	4	5	6	7	8	9
I_1	Pipe 1	IF1	ID1	E	WB1					
I_2	Pipe 2	IF2	ID2	idle	EA1	EA2	WB2			
I_3	Pipe 1		IF1	ID1	idle	EA1	EA2	WB1		
I_4	Pipe 2		IF2	ID2	EM1	EM2	EM3	WB2		
I_5	Pipe 1			IF1	ID1	EL	WB1			
I_6	Pipe 2			IF2	ID2	idle	EM1	EM2	EM3	WB2

Figure 10: In-order issue out-of-order completion

Instructions may have to be stalled only when there is

RAW (idle cycle in I_2) or

WAW (one idle cycle in I_6) or

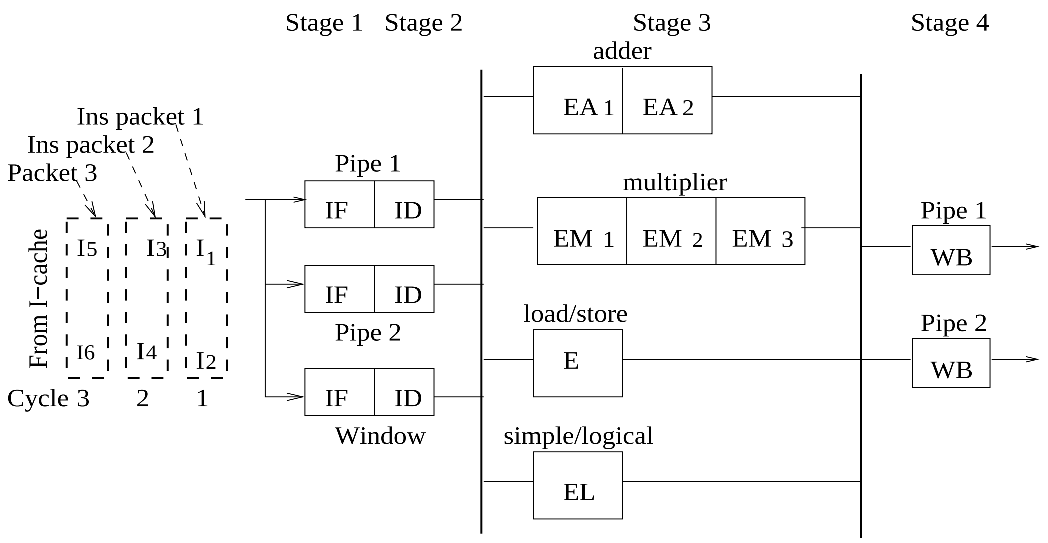
resource conflict (one idle cycle in I_3).

If out-of-order completion policy is followed,

I_5 can be allowed to complete prior to I_3 and I_4 (Figure 10).

I_3 and I_4 are totally independent of I_5.

I_1: $R_1 \leftarrow \text{Memory}(A)$
I_2: $R_2 \leftarrow R_2 + R_1$
I_3: $R_3 \leftarrow R_3 + R_4$
I_4: $R_4 \leftarrow R_4 * R_5$
I_5: $R_6 \leftarrow \neg R_6$
I_6: $R_6 \leftarrow R_6 * R_7$



With lookahead

Time (clock cycle)

Instruction		1	2	3	4	5	6	7	8	9
I_1	Pipe 1	IF1	ID1	E	WB1					
I_2	Pipe 2	IF2	ID2	idle	EA1	EA2	WB2			
I_3	Pipe 1		IF1	ID1	idle	EA1	EA2	WB1		
I_4	Pipe 2		IF2	ID2	EM1	EM2	EM3	WB2		
I_5	Pipe 1			IF1	ID1	EL	WB1			
I_6	Pipe 2			IF2	ID2	idle	EM1	EM2	EM3	WB2

Figure 11: In-order issue out-of-order completion

0.2.3 Out-of-order issue out-of-order-completion

Allows execution of I_{i+1} prior to I_i. It may encounter *anti-dependence* (WAR).

Out-of-order issue is possible if there is provision for lookahead (Figure 12).

In the following, there is an anti-dependence between I_3 and I_4.

$$\begin{array}{ll} I_1: & R_1 \leftarrow \text{Memory}(A) \\ I_2: & R_2 \leftarrow R_2 + R_1 \\ I_3: & R_3 \leftarrow R_3 + R_4 \\ I_4: & R_4 \leftarrow R_4 * R_5 \\ I_5: & R_6 \leftarrow \neg R_6 \\ I_6: & R_6 \leftarrow R_6 * R_7 \end{array}$$

An inst (I_5) can be decoded in advance if it is independent of others (lookahead).

Execution in out-of-order issue out-of-order completion is shown in Figure 12.

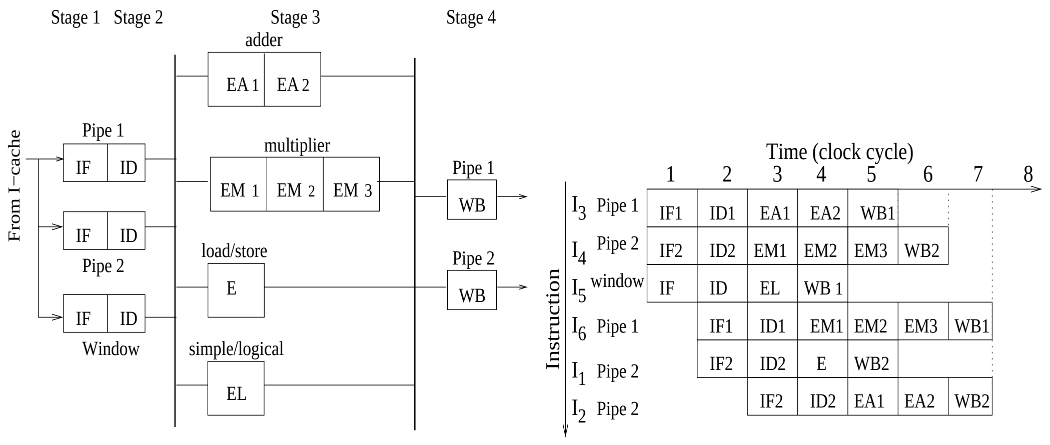


Figure 12: Out-of-order issue out-of-order completion

Six instructions are issued in three cycles.

I_5 is fetched and decoded in the window, while I_3 and I_4 are decoded concurrently.

Execution time is reduced to 7 cycles (there is no idle stage during execution).

As the issue is out of-order, completion is also out of-order.

I_1: $R_1 \leftarrow \text{Memory}(A)$

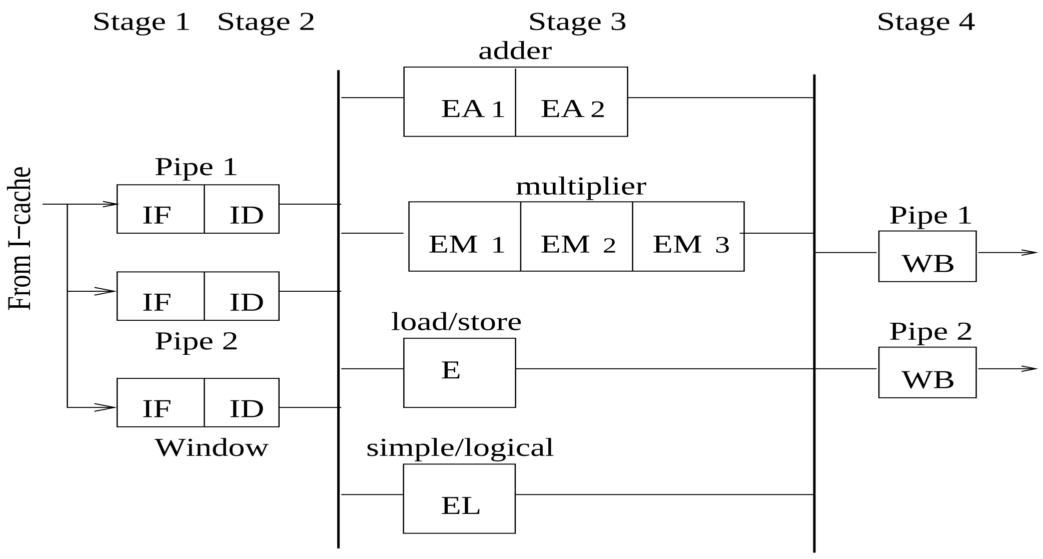
I_2: $R_2 \leftarrow R_2 + R_1$

I_3: $R_3 \leftarrow R_3 + R_4$

I_4: $R_4 \leftarrow R_4 * R_5$

I_5: $R_6 \leftarrow \neg R_6$

I_6: $R_6 \leftarrow R_6 * R_7$



Time (clock cycle)

Instruction		1	2	3	4	5	6	7	8
I_3	Pipe 1	IF1	ID1	EA1	EA2	WB1			
I_4	Pipe 2	IF2	ID2	EM1	EM2	EM3	WB2		
I_5	window	IF	ID	EL	WB 1				
I_6	Pipe 1		IF1	ID1	EM1	EM2	EM3	WB1	
I_1	Pipe 2		IF2	ID2	E	WB2			
I_2	Pipe 2			IF2	ID2	EA1	EA2	WB2	

Figure 13: Out-of-order issue out-of-order completion

Lecture 20-21: October 7, 2020
Computer Architecture and Organization-II
Biplab K Sikdar

ILP -II

Output dependencies/anti-dependencies can be eliminated by *register renaming*.

0.2.4 Register renaming

Processor dynamically allocates additional physical registers

to remove storage conflicts WAR/WAW.

A new register is allocated for every instruction that writes to a register.

Consider the following program segment

$$\begin{aligned}&I_w\text{: } R_1 \leftarrow M(X_1)\\&I_x\text{: } M(X_2) \leftarrow R_1\\&I_y\text{: } R_1 \leftarrow R_2 \times R_3\\&I_z\text{: } R_1 \leftarrow R_4 + R_5\end{aligned}$$

Here, a superscalar processor can not finish execution of I_y until I_x is completed.

There is an anti-dependence (WAR) between I_x and I_y.

Register renaming technique modifies the program to avoid WAR.

Destination register R_1 in I_y (consequently, R_1 in I_z) is renamed as R_6.

$$\begin{aligned}&I_w\text{: } R_1 \leftarrow M(X_1)\\&I_x\text{: } M(X_2) \leftarrow R_1\\&I_y\text{: } R_6 \leftarrow R_2 \times R_3\\&I_z\text{: } R_6 \leftarrow R_4 + R_5\end{aligned}$$

Further, I_z can not be completed before I_y as there is WAW between I_y and I_z.

Both instructions target write to R_6.

This WAW issue is avoided by reallocating outcome of I_z to R_7 (shown below).

$$\begin{aligned}&I_w\text{: } R_1 \leftarrow M(X_1)\\&I_x\text{: } M(X_2) \leftarrow R_1\\&I_y\text{: } R_6 \leftarrow R_2 \times R_3\\&I_z\text{: } R_7 \leftarrow R_4 + R_5.\end{aligned}$$

Pentium: is having 2 integer units and a floating point unit. It follows in-order issue.

Pentium II: allows out-of-order issue. In one cycle, five instructions can be issued.

0.2.5 Scheduling

Pipeline architecture of superscalar processor is shown in Figure 14.

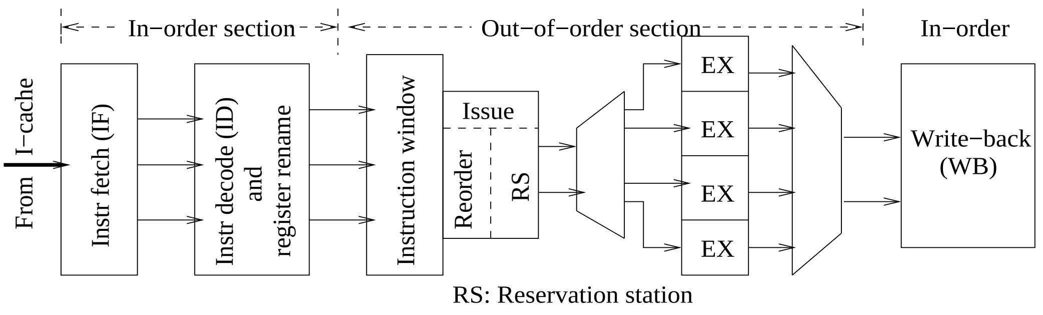


Figure 14: Superscalar architecture

At ID stage operand and destination registers are renamed.

Instructions at instruction window stage are free from WAR and WAW.

Issue block of pipeline resolves data dependence and structural dependence.

Order of instructions, to be scheduled for execution (EX),
    is stored in reorder (in-order or out-of-order) buffer.

Issue block is having reservation stations (buffer) for resolving structural hazards.

0.2.6 Performance of superscalar processing

Performance is evaluated with respect to performance of basescalar processor (m=1).

Time taken to execute N instructions in basescalar processor is

$$T(1,1) = k + N\text{-}1 \text{ base cycles (clock cycles)}$$

Where k is the number of pipeline stages.

Ideal execution time for N instructions in m-issue superscalar processor is

$$T(m,1) = k + \frac{N-m}{m} \text{ base cycles.}$$

After first k cycles, 1^{st} m instructions get executed through m pipelines.

m of the rest (N-m) instructions get executed per cycle through m pipelines.

Ideal speedup of superscalar processor is then

$$S_k(m,1) = \frac{T(1,1)}{T(m,1)} = \frac{m(N+k-1)}{N+m(k-1)}. = \frac{m+\frac{k-1}{N}}{1+\frac{m(k-1)}{N}}.$$

When N tends to ∞, the speedup

$$S_k(m,1) = m.$$

0.3 Superpipelined Processors

Figure 15 shows execution in a superpipelined processor of degree n = 2.

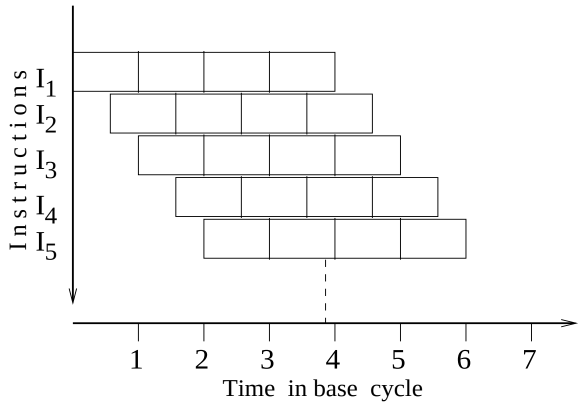


Figure 15: Superpipelining with degree n = 2

Its pipeline cycle time is $\frac{1}{2}$.

In a superpipelined processor of degree n, pipeline cycle time is $\frac{1}{n}$ of base cycle.

Time to execute N instructions in k-stage superpipelined processor of degree n is

$$\mathrm{T}(1,n) = k + \frac{N-1}{n} \text{ base cycles}$$

Speedup of superpipelined processor, with degree n, over base scalar processor is

$$\mathrm{S}_k(1,n) = \frac{T(1,1)}{T(1,n)} = \frac{k+N-1}{k+\frac{N-1}{n}} = \frac{n(k+N-1)}{nk+N-1} = \frac{\frac{nk}{N}+n-\frac{n}{N}}{\frac{nk}{N}+1-\frac{1}{N}}$$

When N tends to ∞, the speed up

$$\mathrm{S}_k(1,n) = n.$$

In Figure 15, the speed up is $\mathrm{S}_k(1,2) = 2$.

0.4 Superpipelined Superscalar Design

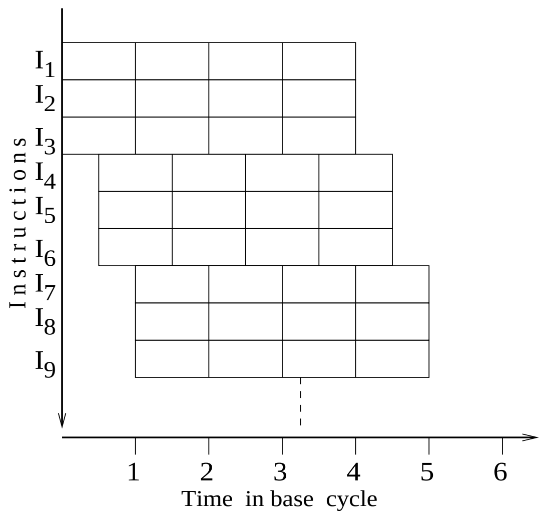


Figure 16: Superscalar superpipelined execution with m = 3, n = 2

Superpipeline and superscalar technology are combined.

Figure 16 is of degree(m=3,n=2).

$m \times n$ instructions are get executed in each cycle.

To exploit full parallelism, we require to find mn independent instructions.

Time, in terms of base cycles, needed to execute N independent instructions

$$\text{T}(m,n) = k + \frac{N-m}{mn} \text{ base cycles}$$

Thus speedup over the base scalar processor is

$$\text{S}_k(m,n) = \frac{T(1,1)}{T(m,n)} = \frac{k+N-1}{k+\frac{N-m}{mn}},$$

That is,

$$\text{S}_k(m,n) \simeq mn, \text{ as N tends to } \infty.$$

For example of Figure 16, m = 3, n = 2. That is, 3×2 = 6.

Major limitation in superscalar processing is the scheduling of instructions in pipe.

0.5 Very Long Instruction Word

Very long instruction word (VLIW) eliminates complicated instruction scheduling.

It realizes static superscalar processing.

Execution of instructions by an ideal VLIW processor is explained in Figure 17.

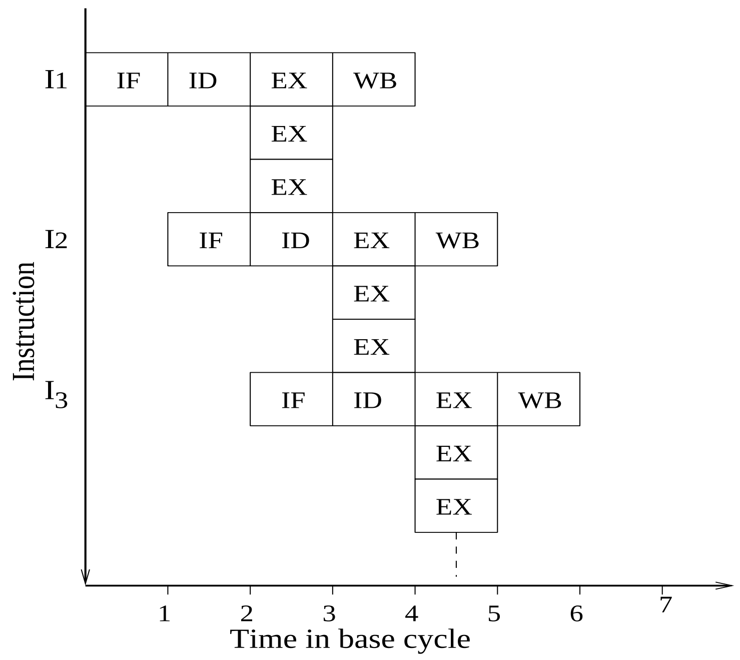


Figure 17: VLIW execution for degree m = 3

Each instruction consists of multiple independent parallel operations.

For degree m = 3, effective CPI is $\frac{1}{m}$ = 0.33.

VLIW instruction format/word - normally instruction length is of 128 to 1024 bits.

VLIW is generalized form of - horizontal micro-coding and superscalar processing.

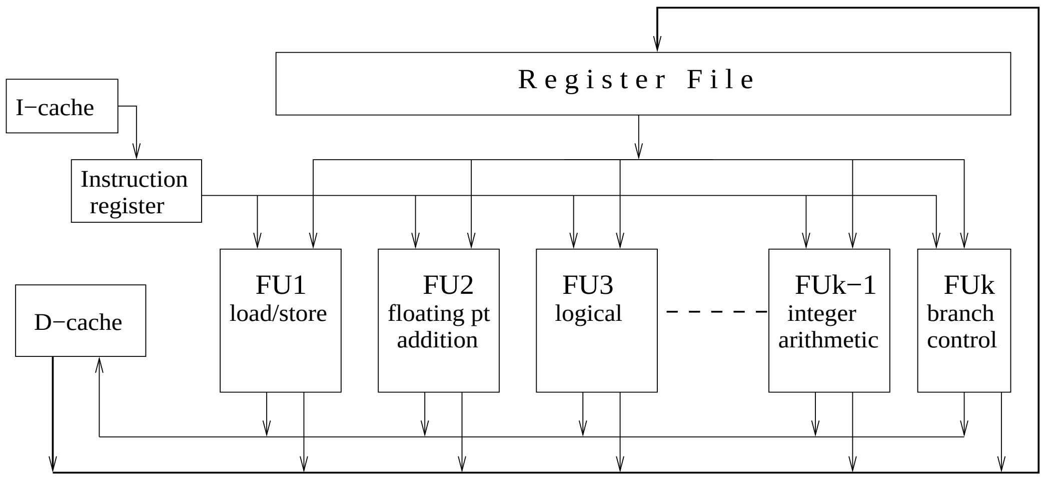


Figure 18: VLIW processor architecture

VLIW processor (Figure 18) has central controller.

It issues long instruction word in a cycle. Instruction is then decoded.

Operations within instruction (Figure 19) are dispatched to respective FUs.

That is, load/store to FU_1, integer arithmetic operations to FU_{k-1}, and so on.

Operations are simultaneously executed by the functional units.

FUs are connected through a global shared register file (Figure 18).

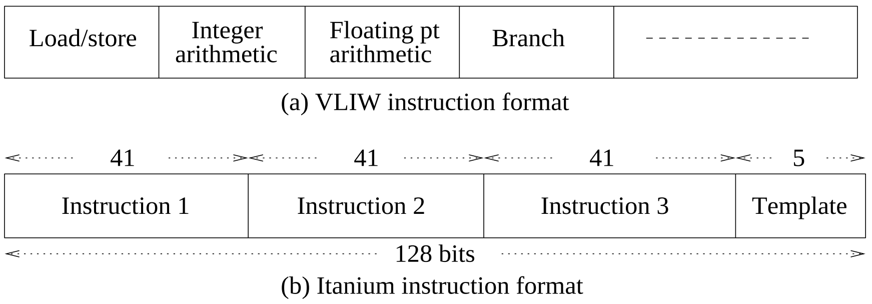


Figure 19: VLIW instruction format

Program in conventional instruction words is compacted to form VLIW instructions.

Code compaction must be done by a compiler which can predict branch outcome.

Example 0.1 Consider the program segment in 3-address assembly code.

```
I1:  T1  = A + B
I2:  T2  = C - D
I3:   F  = T1/T2
I4:   C  = C-1
```

To schedule operations in VLIW instructions, compiler performs dataflow analysis.

It derives an operation precedence graph from a portion of program.

Independent operations can be scheduled to execute concurrently.

T1, T2 and T3 are independent and executed concurrently if resource is available.

For a 3-wide VLIW, VLIW instructions can be set as in Figure 20.

i:	ADD T1, A, B	SUB T2, C, D	SUB C, C, 1
i+1:	DIV F, T1, T2	NOP	NOP

Figure 20: VLIW instruction

Success of VLIW processor depends heavily on efficiency in code compaction.

This design is totally incompatible with conventional general-purpose processors.

Performs well in scientific applications
where program behavior (branch prediction) is more predictable.

VLIW processor is like a superscalar processor with following differences.

1. Code density of superscalar processor is better

   When available ILP is less than that exploitable by VLIW processor.

   As fixed VLIW format includes bits for non-executable operations (NOP).

   While superscalar processor issues only executable instructions.

   Follow the figure

i:	ADD T1, A, B	SUB T2, C, D	SUB C, C, 1
i+1:	DIV F, T1, T2	NOP	NOP

Figure 21: VLIW instruction

   Compiler of VLIW places NOP instruction in the slots.

2. VLIW - Instruction parallelism and data movement:

   are completely specified at compile time.

   It reduces hardware complexity over a superscalar implementation.

   Run-time resource scheduling and synchronization are also eliminated in VLIW.

   But it requires a complex compiler.

INTEL's first VLIW processor, Itanium

Instruction format is shown in Figure 22.

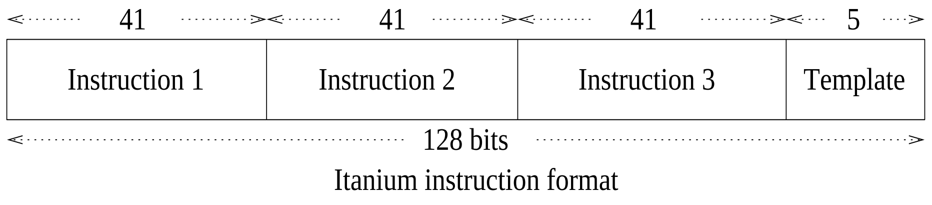


Figure 22: VLIW Itanium processor instruction format

It has 128-bit instructions.

5-bit template specifies instruction/operation type in the instruction bundle.

It points to instructions/operations that can be executed concurrently.

This information is from compiler for CPU and explicit.

A 41-bit instruction contains

a) 14-bit opcode and flags,

b) 6-bit for predicate specification (refers to conditional branches and a set of 64 predicate registers).

   Predicate = 1 means instruction is committed; otherwise discarded.

c) 21 bits for 3-operands.

   To specify an operand, 7-bits are required as 128 registers are in Itanium.

Simple branch prediction may loose efficiency of a superscalar/VLIW processor.

This is resolved through speculative execution.