

# SUPERSCALAR PROCESSORS

1. What is a Superscalar Architecture?
2. Superpipelining
3. Features of Superscalar Architectures
4. Data Dependencies
5. Policies for Parallel Instruction Execution
6. Register Renaming

## What is a Superscalar Architecture?

- A superscalar architecture is one in which several instructions can be initiated *simultaneously* and executed *independently*.
- Pipelining allows several instructions to be executed at the same time, but they have to be in *different* pipeline stages at a given moment.
- Superscalar architectures include all features of pipelining but, in addition, there can be several instructions *executing simultaneously in the same* pipeline stage.  
They have the ability to initiate multiple instructions *during the same clock cycle*.

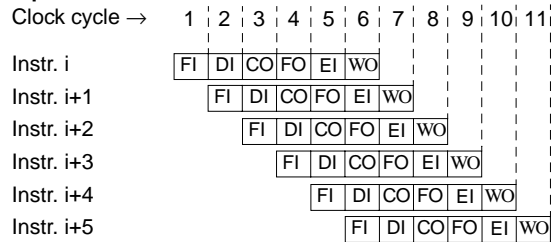
There are two typical approaches today, in order to improve performance:

1. Superpipelining
2. Superscalar

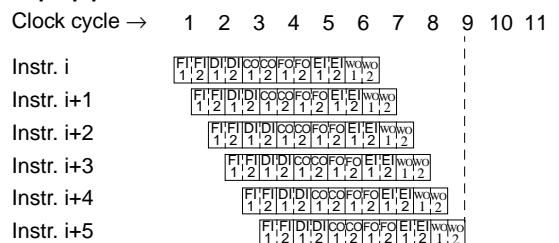
## Superpipelining

- Superpipelining is based on dividing the stages of a pipeline into substages and thus increasing the number of instructions which are supported by the pipeline at a given moment.
- By dividing each stage into two, the clock cycle period  $\tau$  will be reduced to the half,  $\tau/2$ ; hence, at the maximum capacity, the pipeline produces a result every  $\tau/2$  s.
- For a given architecture and the corresponding instruction set there is an optimal number of pipeline stages; *increasing the number of stages over this limit reduces the overall performance.*
- *A solution to further improve speed is the superscalar architecture.*

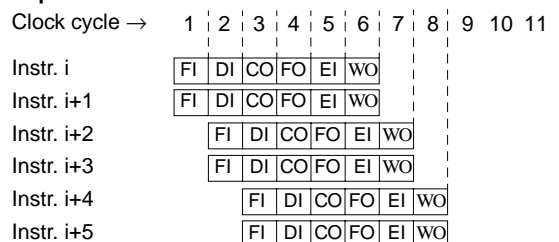
## Pipelined execution



## Superpipelined execution



## Superscalar execution

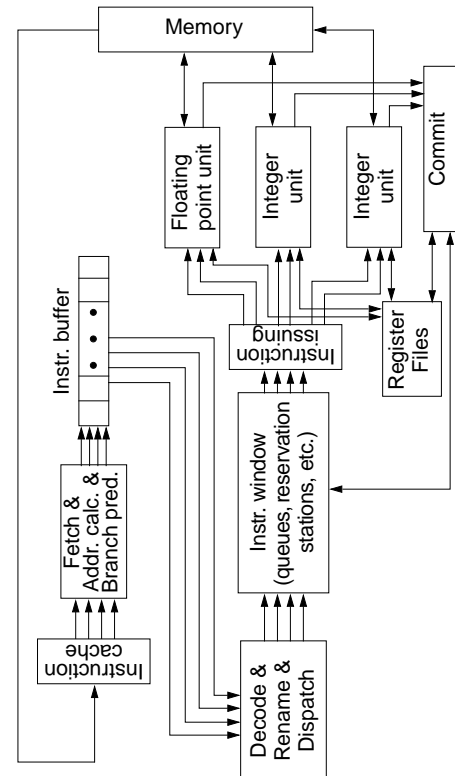


## Superscalar Architectures

- Superscalar architectures allow several instructions to be issued and completed per clock cycle.
- A superscalar architecture consists of a number of pipelines that are working in parallel.
- Depending on the number and kind of parallel units available, a certain number of instructions can be executed in parallel.
- In the following example a floating point and two integer operations can be issued and executed simultaneously; each unit is pipelined and can execute several operations in different pipeline stages.



## Superscalar Architectures (cont'd)



## Limitations on Parallel Execution

- The situations which prevent instructions to be executed in parallel by a superscalar architecture are very similar to those which prevent an efficient execution on any pipelined architecture (see *pipeline hazards* - lectures 3, 4).
- The consequences of these situations on superscalar architectures are more severe than those on simple pipelines, because the potential of parallelism in superscalars is greater and, thus, a greater opportunity is lost.



## Limitations on Parallel Execution (cont'd)

- Three categories of limitations have to be considered:
  1. Resource conflicts:
    - They occur if two or more instructions compete for the same resource (register, memory, functional unit) at the same time; they are similar to *structural hazards* discussed with pipelines. Introducing several parallel pipelined units, superscalar architectures try to reduce a part of possible resource conflicts.
  2. Control (procedural) dependency:
    - The presence of branches creates major problems in assuring an optimal parallelism. How to reduce branch penalties has been discussed in lectures 7&8.
    - If instructions are of variable length, they cannot be fetched and issued in parallel; an instruction has to be decoded in order to identify the following one and to fetch it  $\Rightarrow$  superscalar techniques are efficiently applicable to RISCs, with fixed instruction length and format.
  3. Data conflicts:
    - Data conflicts are produced by *data dependencies* between instructions in the program. Because superscalar architectures provide a great liberty in the order in which instructions can be issued and completed, data dependencies have to be considered with much attention.



### Limitations on Parallel Execution (cont'd)

Instructions have to be issued as much as possible in parallel.



- Superscalar architectures exploit the potential of instruction level parallelism present in the program.
- An important feature of modern superscalar architectures is dynamic instruction scheduling:
  - instructions are issued for execution dynamically, in parallel and *out of order*.
  - *out of order issuing*: instructions are issued independent of their sequential order, based only on dependencies and availability of resources.

Results must be identical with those produced by strictly sequential execution.



- Data dependencies have to be considered carefully



### Limitations on Parallel Execution (cont'd)

- Because of data dependencies, only some part of the instructions are potential subjects for parallel execution.



- In order to find instructions to be issued in parallel, the processor has to select from a sufficiently large instruction sequence.



- A large *window of execution* is needed



### Window of Execution

- Window of execution:

The set of instructions that is considered for execution at a certain moment. Any instruction in the window can be issued for parallel execution, subject to data dependencies and resource constraints.

- The number of instructions in the window should be as large as possible.  
Problems:
  - Capacity to fetch instructions at a high rate
  - The problem of branches



### Window of Execution (cont'd)

```
for (i=0; i<last; i++) {
  if (a[i] > a[i+1]) {
    temp = a[i];
    a[i] = a[i+1];
    a[i+1] = temp;
    change++;
  }
}
```

r7: address of current element (a[i])

r3: address for access to a[i], a[i+1]

r5: *change*; r4: *last*; r6: *i*

```
L2  move  r3,r7
     lw   r8,(r3)  r8 ← a[i]
     add  r3,r3,4
     lw   r9,(r3)  r9 ← a[i+1]
     ble  r8,r9,L3 } basic block 1

     move r3,r7
     sw   r9,(r3)  a[i] ← r9
     add  r3,r3,4
     sw   r8,(r3)  a[i+1] ← r8
     add  r5,r5,1  change++
     } basic block 2

L3  add  r6,r6,1  i++
     add  r7,r7,4
     blt  r6,r4,L2 } basic block 3
```



### Window of Execution (cont'd)

- The window of execution is extended over basic block borders by *branch prediction*



speculative execution

- With speculative execution, instructions of the predicted path are entered into the window of execution.

Instructions from the predicted path are executed tentatively.  
If the prediction turns out to be correct the state change produced by these instructions will become permanent and visible (the instructions *commit*); if not, all effects are removed.



### Data Dependencies

- All instructions in the window of execution may begin execution, subject to *data dependence* (and resource) *constraints*.

- Three types of data dependencies can be identified:

1. True data dependency

2. Output dependency

3. Antidependency

} artificial dependencies



### True Data Dependency

- True data dependency* exists when the output of one instruction is required as an input to a subsequent instruction:

```
MUL  R4,R3,R1  R4 ← R3 * R1
```

-----

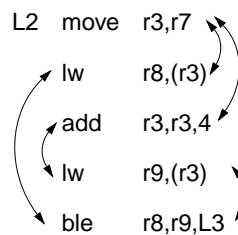
```
ADD  R2,R4,R5  R2 ← R4 + R5
```

- True data dependencies are intrinsic features of the user's program. They *cannot be eliminated* by compiler or hardware techniques.
- True data dependencies *have to be detected and treated*: the addition above cannot be executed before the result of the multiplication is available.
  - The simplest solution is to stall the adder until the multiplier has finished.
  - In order to avoid the adder to be stalled, the compiler or hardware can find other instructions which can be executed by the adder until the result of the multiplication is available.



### True Data Dependency

For the example on slide 12:



### Output Dependency

- An *output dependency* exists if two instructions are writing into the same location; if the second instruction writes before the first one, an error occurs:

```
MUL  R4,R3,R1  R4 ← R3 * R1
-----
ADD  R4,R2,R5  R4 ← R2 + R5
```

For the example on slide 12:

```
L2  move  r3,r7
     lw    r8,(r3)
     add   r3,r3,4
     lw    r9,(r3)
     ble   r8,r9,L3
```

### Antidependency

- An *antidependency* exists if an instruction uses a location as an operand while a following one is writing into that location; if the first one is still using the location when the second one writes into it, an error occurs:

```
MUL  R4,R3,R1  R4 ← R3 * R1
-----
ADD  R3,R2,R5  R3 ← R2 + R5
```

For the example on slide 12:

```
L2  move  r3,r7
     lw    r8,(r3)
     add   r3,r3,4
     lw    r9,(r3)
     ble   r8,r9,L3
```

### The Nature of Output Dependency and Antidependency

- Output dependencies and antidependencies are not intrinsic features of the executed program; they are not *real* data dependencies but *storage conflicts*.
- Output dependencies and antidependencies are only the consequence of the manner in which the programmer or the compiler are using registers (or memory locations). They are produced by the competition of several instructions for the same register.
- In the previous examples the conflicts are produced only because:
  - the output dependency: R4 is used by both instructions to store the result;
  - the antidependency: R3 is used by the second instruction to store the result;
- The examples could be written without dependencies *by using additional registers*:

```
MUL  R4,R3,R1  R4 ← R3 * R1
-----
ADD  R7,R2,R5  R7 ← R2 + R5
```

and

```
MUL  R4,R3,R1  R4 ← R3 * R1
-----
ADD  R6,R2,R5  R6 ← R2 + R5
```

### The Nature of Output Dependency and Antidependency (cont'd)

Example from slide 12:

```
L2  move  r3,r7
     lw    r8,(r3)
     add   r3,r3,4
     lw    r9,(r3)
     ble   r8,r9,L3
```



```
L2  move  R1,r7
     lw    r8,(R1)
     add   R2,R1,4
     lw    r9,(R2)
     ble   r8,r9,L3
```

### Policies for Parallel Instruction Execution

- The ability of a superscalar processor to execute instructions in parallel is determined by:
  - the number and nature of parallel pipelines (this determines the number and nature of instructions that can be fetched and executed at the same time);
  - the mechanism that the processor uses to find independent instructions (instructions that can be executed in parallel).
- The policies used for instruction execution are characterized by the following two factors:
  - the order in which instructions are issued for execution;
  - the order in which instructions are completed (they write results into registers and memory locations).

### Policies for Parallel Instruction Execution (cont'd)

- The simplest policy is to execute and complete instructions in their sequential order. This, however, gives little chances to find instructions which can be executed in parallel.
- In order to improve parallelism the processor has to look ahead and try to find independent instructions to execute in parallel.



Instructions will be executed in an order different from the strictly sequential one, **with the restriction that the result must be correct.**

- Execution policies:
  - In-order issue with in-order completion.
  - In-order issue with out-of-order completion.
  - Out-of-order issue with out-of-order completion.

### Policies for Parallel Instruction Execution (cont'd)

We consider the following instruction sequence:

- I1: ADDF R12,R13,R14  $R12 \leftarrow R13 + R14$  (float. pnt.)
- I2: ADD R1,R8,R9  $R1 \leftarrow R8 + R9$
- I3: MUL R4,R2,R3  $R4 \leftarrow R2 * R3$
- I4: MUL R5,R6,R7  $R5 \leftarrow R6 * R7$
- I5: ADD R10,R5,R7  $R10 \leftarrow R5 + R7$
- I6: ADD R11,R2,R3  $R11 \leftarrow R2 + R3$

- I1 requires two cycles to execute;
- I3 and I4 are in conflict for the same functional unit;
- I5 depends on the value produced by I4 (we have a *true data dependency* between I4 and I5);
- I2, I5 and I6 are in conflict for the same functional unit;

### In-Order Issue with In-Order Completion

- Instructions are issued in the exact order that would correspond to sequential execution; results are written (completion) in the same order.



- An instruction cannot be issued before the previous one has been issued;
- An instruction completes only after the previous one has completed.

Decode/ Issue		Execute		Writeback/ Complete		Cycle
I1	I2					1
I3	I4	I1	I2			2
I5	I6	I1				3
			I3	I1	I2	4
			I4	I3		5
			I5	I4		6
			I6	I5		7
				I6		8

- To guarantee in-order completion, instruction issuing stalls when there is a conflict *and when the unit requires more than one cycle to execute;*

### In-Order Issue with In-Order Completion (cont'd)

- The processor detects and handles (by stalling) true data dependencies and resource conflicts.
- As instructions are issued and completed in their strict order, the resulting parallelism is very much dependent on the way the program is written/compiled.



If I3 and I6 switch position, the pairs I6-I4 and I5-I3 can be executed in parallel (see following slide).

- With superscalar processors we are interested in techniques which are not compiler based but allow the hardware alone to detect instructions which can be executed in parallel and to issue them.**

### In-Order Issue with In-Order Completion (cont'd)

If the compiler generates this sequence:

```

I1: ADDF R12,R13,R14 R12 ← R13 + R14 (float. pnt.)
I2: ADD R1,R8,R9 R1 ← R8 + R9
I6: ADD R11,R2,R3 R11 ← R2 + R3
I4: MUL R5,R6,R7 R5 ← R6 * R7
I5: ADD R10,R5,R7 R10 ← R5 + R7
I3: MUL R4,R2,R3 R4 ← R2 * R3
    
```

I6-I4 and I5-I3 could be executed in parallel

Decode/ Issue		Execute		Writeback/ Complete		Cycle	
I1	I2					1	
I6	I4	I1	I2			2	
I5	I3	I1				3	
			I6	I4	I1	I2	4
			I5	I3	I6	I4	5
					I5	I3	6
							7
							8

- The sequence needs only 6 cycles instead of 8.

### In-Order Issue with In-Order Completion (cont'd)

- With *in-order issue* & *in-order completion* the processor has not to bother about output-dependency and antidependency! **It has only to detect true data dependencies.**

**No one of the two dependencies will be violated if instructions are issued/completed in-order:**

output dependency

```

MUL R4,R3,R1 R4 ← R3 * R1
-----
ADD R4,R2,R5 R4 ← R2 + R5
    
```

Antidependency

```

MUL R4,R3,R1 R4 ← R3 * R1
-----
ADD R3,R2,R5 R3 ← R2 + R5
    
```

### Out-of-Order Issue with Out-of-Order Completion

- With in-order issue, no new instruction can be issued when the processor has detected a conflict and is stalled, until after the conflict has been resolved.



The processor is not allowed to *look ahead* for further instructions, which could be executed in parallel with the current ones.

- Out-of-order issue tries to resolve the above problem. Taking the set of decoded instructions the processor looks ahead and issues any instruction, in any order, as long as the program execution is correct.

### Out-of-Order Issue with Out-of-Order Completion (cont'd)

We consider the instruction sequence in slide 15.

- I6 can be now issued before I5 and in parallel with I4; the sequence takes only 6 cycles (compared to 8 if we have in-order issue&in-order completion and to 7 with in-order issue&out-of-order completion).

Decode/ Issue	Execute	Writeback/ Complete	Cycle
I1 I2			1
I3 I4	I1 I2		2
I5 I6	I1 I3	I2	3
	I6 I4	I1 I3	4
	I5	I4 I6	5
		I5	6
			7
			8

### Out-of-Order Issue with Out-of-Order Completion (cont'd)

- With *out-of-order issue&out-of-order completion* the processor has to bother about true data dependency and **both about output-dependency and antidependency!**

Output dependency can be violated (the addition completes before the multiplication):

```
MUL  R4,R3,R1  R4 ← R3 * R1
-----
ADD  R4,R2,R5  R4 ← R2 + R5
```

Antidependency can be violated (the operand in R3 is used after it has been over-written):

```
MUL  R4,R3,R1  R4 ← R3 * R1
-----
ADD  R3,R2,R5  R3 ← R2 + R5
```

### Register Renaming

- Output dependencies and antidependencies can be treated similarly to true data dependencies as normal conflicts. Such conflicts are solved by delaying the execution of a certain instruction until it can be executed.
- Parallelism could be improved by *eliminating* output dependencies and antidependencies, **which are not real data dependencies** (see slide 11).
- Output dependencies and antidependencies can be eliminated by automatically allocating new registers to values, when such a dependency has been detected. This technique is called **register renaming**.

The *output dependency* is eliminated by allocating, for example, R6 to the value R2+R5:

```
MUL  R4,R3,R1  R4 ← R3 * R1
-----
ADD  R4,R2,R5  R4 ← R2 + R5
(ADD  R6,R2,R5  R6 ← R2 + R5)
```

The same is true for the *antidependency* below:

```
MUL  R4,R3,R1  R4 ← R3 * R1
-----
ADD  R3,R2,R5  R3 ← R2 + R5
(ADD  R6,R2,R5  R6 ← R2 + R5)
```

### Final Comments

- The following main techniques are characteristic for superscalar processors:
  - additional pipelined units which are working in parallel;
  - out-of-order issue&out-of-order completion;
  - register renaming.
- All of the above techniques are aimed to enhance performance.
- Experiments have shown:
  - without the other techniques, only adding additional units is not efficient;
  - out-of-order issue is extremely important; it allows to look ahead for independent instructions;
  - register renaming can improve performance with more than 30%; in this case performance is limited only by *true dependencies*.
  - it is important to provide a fetching/decoding capacity so that the window of execution is sufficiently large.

## Some Architectures

### PowerPC 604

- six independent execution units:
  - Branch execution unit
  - Load/Store unit
  - 3 Integer units
  - Floating-point unit
- in-order issue
- register renaming

### Power PC 620

- provides in addition to the 604 *out-of-order issue*

### Pentium

- three independent execution units:
  - 2 Integer units
  - Floating point unit
- in-order issue

### Pentium II

- provides in addition to the Pentium *out-of-order issue*
- five instructions can be issued in one cycle

