Superscalar Microprocessors

1. Superscalar Processing
2. Main Features of Superscalar Architectures
3. Data Dependencies
4. Renaming
5. Microarchitectural View
6. Out-of-Order Issuing
7. Examples of Superscalar Microprocessors

What is Superscalar Processing?

☞ Superscalar processing is the ability to initiate *multiple instructions during the same clock cycle*.

☞ 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.

Superscalar vs. Pipelined

☞ With classical pipelining, several instructions can be in execution concurrently, however one single instruction is initiated at a time.

⇓

Initiation rate is *one instruction per cycle*.

☞ Superscalar processors initiate *several instructions per cycle*.

- Superscalar architectures make extensive use of pipelining

Superscalar vs. Pipelined (cont'd)

Pipelined execution

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11
Instr. i	FI	DI	CO	FO	EI	WO					
Instr. i+1		FI	DI	CO	FO	EI	WO				
Instr. i+2			FI	DI	CO	FO	EI	WO			
Instr. i+3				FI	DI	CO	FO	EI	WO		
Instr. i+4					FI	DI	CO	FO	EI	WO	
Instr. i+5						FI	DI	CO	FO	EI	WO

Superscalar execution

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11
Instr. i	FI	DI	CO	FO	EI	WO					
Instr. i+1	FI	DI	CO	FO	EI	WO					
Instr. i+2		FI	DI	CO	FO	EI	WO				
Instr. i+3		FI	DI	CO	FO	EI	WO				
Instr. i+4			FI	DI	CO	FO	EI	WO			
Instr. i+5			FI	DI	CO	FO	EI	WO			

Problems

☞ The executed program is based on a sequential execution model.

- Results must be identical with those produced by strictly sequential execution.
- At the time of an interrupt, a *precise state* of the machine (architecturally visible registers and memory) has to be available to be stored in order to resume execution from the interrupted instruction.
  - Precise state: the state that would be present if the sequential execution model was followed and processing was stopped precisely at the interrupted instruction.

⇩

The outward appearance of sequential execution has to be kept.

Problems (cont'd)

To be solved:

- determination of dependence relationships between instructions and possible elimination of dependencies;
- strategies to determine when an operation is ready for execution;
- techniques to pass values from one operation to another (including results of operations which are not yet *committed*);
- strategies for instruction commitment, so that the appearance of sequential execution is maintained.

Features of Superscalar Architectures

- Simultaneous fetching of multiple instructions with branch prediction and fetching beyond conditional branches (speculative execution).
- Capability to determine true dependencies; eliminating “artificial” dependencies.
- Parallel issuing of multiple instructions based on dependencies and resources.
- Availability of resources for parallel execution of instructions: multiple pipelined functional units, memories capable of servicing multiple references.
- Methods for committing process state in correct order.

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.
- 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                        } block 1
     lw     r9,(r3)      r9 ← a[i+1]       }
     ble    r8,r9,L3                       }

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

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


Window of Execution (cont'd)

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

- static branch prediction
- dynamic branch prediction (history tables)

$\Downarrow$

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 constraints*.

- true dependence (read-after-write: RAW)
- antidependence (write-after-read: WAR)
- output dependence (write-after-write: WAW)

artificial dependencies

True Dependence

☞ One instruction is consuming the value produced by another one.

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

- True dependence is the result of the intrinsic data-flow of the algorithm.
- It imposes a partial order of the instructions.

Antidependence

☞ An instruction writes into a storage location which is read by previous instructions.

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


Output Dependence

☞ Multiple instructions update the same location.

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

Artificial Dependencies

☞ Output dependence and antidependence are not intrinsic features of the algorithm but consequences of the programme coding.

⇩

They can be removed by *renaming*

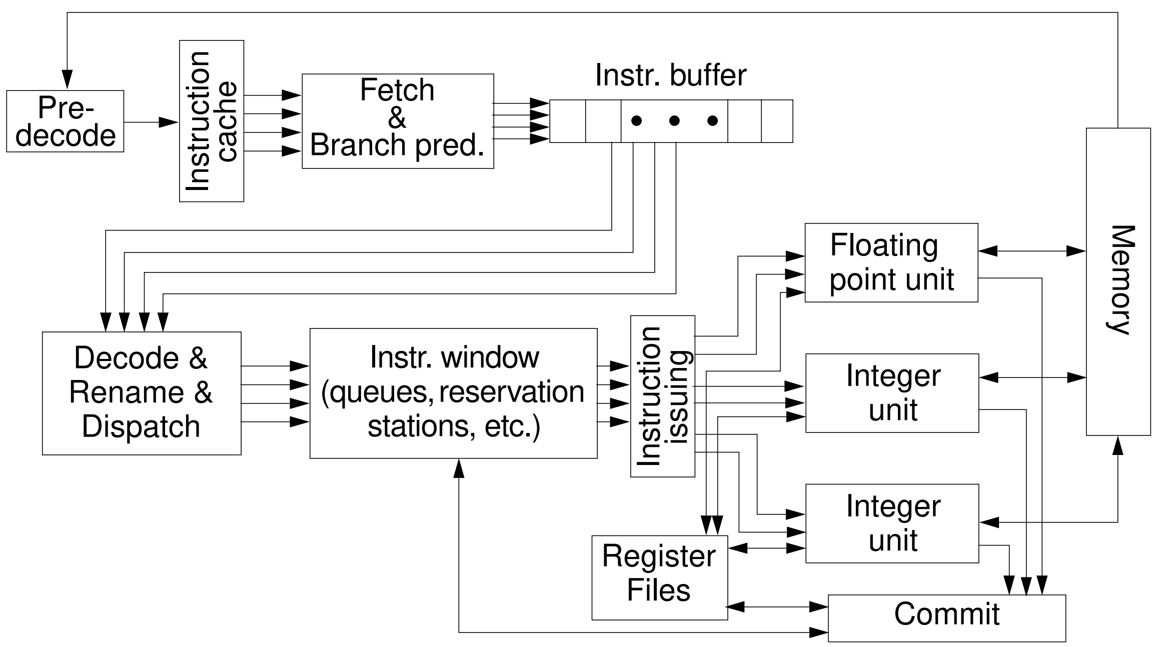
```
L2  move  r3,r7
    lw    r8,(r3)
    add   r3,r3,4
    lw    r9,(r3)
    ble   r8,r9,L3
```

rename *r3* ⇨

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

WAW

Microarchitectural View



Fetch, Branch Prediction, Predecode

☞ Very important: Reduce latency and increase bandwidth of the instruction fetch process. This is needed in order to sustain execution of multiple instructions per cycle.

☞ Separate instruction cache.

☞ Instruction buffer helps smooth out instruction fetch irregularities caused by cache miss and branches.

☞ Branch prediction is extremely important to provide sufficient flow of instructions and sufficiently large instruction windows.

☞ In order to go on with instruction fetching, prediction and address calculation has to be performed "in advance" ⇒ *predecode*.

Decode, Rename, Dispatch

☞ Instructions are removed from the instruction buffer, are decoded, checked for dependencies and artificial dependencies are removed by *renaming*.

☞ For renaming, register names given by the programmer are considered as "logical registers". When a logical register gets a value, a physical location (register) is assigned to it; all subsequent references to that logical register are replaced by links to the physical location (based on a *mapping table*).

☞ The job of dispatching is to set up *execution tuples* for each instruction, and to place them into queues or reservation stations for later issuing. Execution tuple:

- operation to be executed
- identity of storage elements where operands (will) reside
- identity of location were result will be placed.

Instruction Issuing

☞ Instruction issuing decides on which instructions can be executed. The goal is maximum parallelism. The restrictions are:
- availability of data
- availability of resources (functional units, path to memory, internal paths, etc.

☞ Execution tuples are dispatched to reservation stations ⇒
- Available operands are loaded from the register files;
- the other operands are loaded as soon as they are produced by the respective instructions;
- as soon as all operands are there and resources are available, the instruction is issued.

Instruction Issuing (cont'd)

This is the sequence from slide 46, as issued fro parallel execution on: two integer units, one path to the memory, one branch unit.

Integer unit 1	Integer unit 2	Memory path	Branch unit
(1) move R1,r7			
(3) add R2,R1,4		(2) lw r8,(R1)	
(6) move R3,r7		(4) lw r9,(R2)	
(8) add R4,R3,4			
(10) add r5,r5,1	(11) add r6,r6,1	(7) sw r9,(R3)	(5) ble r8,r9,L3
(12) add r7,r7,4		(9) sw r8,(R4)	(13) blt r6,r4,L2

☞ Memory access takes two cycles (delay of operations 7 and 5).

☞ It is predicted that branch *ble* is not taken.

☞ Only renaming of register *r3* is shown.

☞ Observe **out-of-order execution**.

Instruction Commit

☞ In the commit phase, the effects of the instruction are allowed to modify the externally visible (logical) process state.

☞ The purpose is to implement the appearance of the sequential execution model.

☞ At commit, the result (effect) of an instruction is moved into the architectural (visible) register file; if the committed instruction is a *store*, buffered data is written to memory.

☞ An instruction can be committed if:
- All of its predecessors, corresponding to sequential model, are committed.
- If the block to which it belongs is control dependent on a conditional branch, the respective branch has been validated.

MIPS R10000

- Branch prediction is based on a prediction table with 512 entries. Two-bits based dynamic branch prediction.
- Instruction are dispatched into three instruction queues:
  - to memory
  - to integer units
  - to floating point units.

  Each queue acts similar to a reservation station.
- Five functional units are available for parallel out-of order execution:
  - address adder
  - two integer ALUs
  - a floating point multiplier/divider/square rooter
  - floating point adder

DEC Alpha 21164

- Instructions are issued in-order. However, control logic is simplified, which allows a high clock rate.
- Branch prediction is based on a prediction table.
  Two-bits based dynamic branch prediction.
- Four functional units:
  - two integer ALUs
  - floating point adder
  - floating point multiplier

Pentium Pro

- The Pentium is a CISC. Superscalar execution is at the level of the microinstructions.
- Instruction fetch is complicated by the fact that instructions are of variable length.
- Instructions are decoded, converted to microoperations which are buffered and executed in superscalar mode.
- Branch prediction is based on a prediction table.
- Out-of-order execution based on a unique reservation station and five functional units:
  - two integer units
  - two floating point units
  - one memory interface unit

AMD K5/6

- AMD implements the x86 instruction set (CISC).
- After decoding, x86 instructions are converted to RISC-like Operations (ROPs). They are executed on a RISC core which is at the heart of the AMD architecture (RISC86 microarchitecture).
- Instructions are dispatched to reservation stations which are partitioned among the functional units.
- Branch prediction is based on a prediction table.
- Out-of-order execution based on six functional units:
  - two integer units
  - one floating point units
  - two load/store units
  - a branch unit

Summary

- Superscalar processors initiate multiple instructions during the same clock cycle.
- Instructions are issued dynamically, out-of-order. Maximal parallelism is the goal, constrained by data dependencies and availability of resources.
- Results must be identical with those produced by strictly sequential execution. Commitment of instructions is performed so that the appearance of sequential execution is kept.
- In order to enhance the potential of parallelism, speculative execution is performed. Speculative execution is based on branch prediction.
- Data dependencies have to be detected. Artificial dependencies are removed by renaming.