ADVANCED COMPUTER ARCHITECTURE

Khoa Khoa học và Kỹ thuật Máy tính
BM Kỹ thuật Máy tính

Trần Ngọc Thịnh
http://www.cse.hcmut.edu.vn/~tnthinh

SUPERSCALAR AND VLIW PROCESSORS

Outline

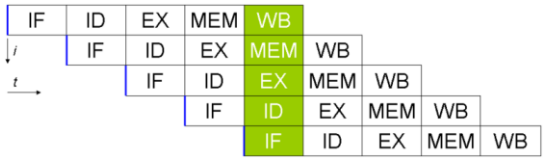
- What is a Superscalar Architecture?
- Features of Superscalar Architectures
- Data Dependencies
- Policies for Parallel Instruction Execution
- Register Renaming
- VLIW Processors

What is a Superscalar Architecture?

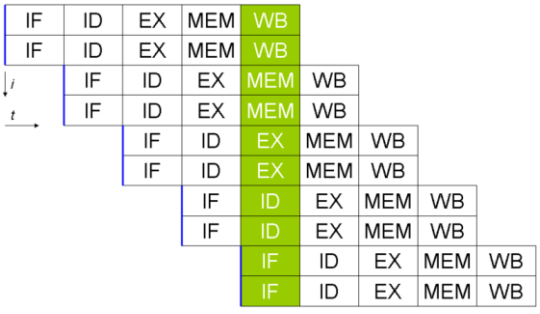
- A superscalar architecture is one in which several instructions can be initiated simultaneously and executed independently.
- Pipelining allows several instructions to be executed at the same time, but they have to be in different pipeline stages at a given moment.
- Superscalar architectures include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

What is a Superscalar Architecture?

- Pipelined execution



- **Superscalar execution**



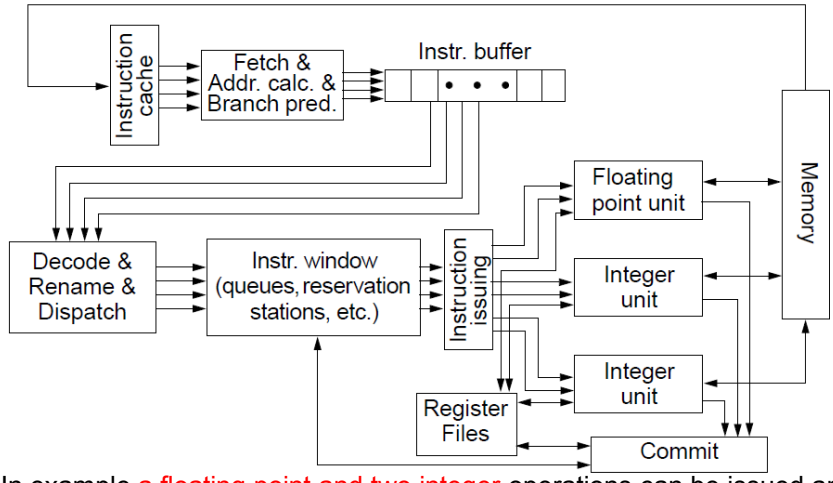
5

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.

6

Superscalar Architectures



- In 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.

7

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.
- 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.

8

Limitations on Parallel Execution

- 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.
   - 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 Þsuperscalar 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.

Data Dependencies

- Three types of data dependencies can be identified:
  1. True data dependency
  2. Output dependency
  3. Antidependency

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 unti 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.

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

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
```

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
```

Policies for Parallel Instruction Execution

- The ability of a superscalar processor to execute instructions in parallel is determined by:
  1. 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);
  2. 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:
  1. the order in which instructions are issued for execution;
  2. the order in which instructions are completed (they write results into registers and memory locations).

Policies for Parallel Instruction Execution

- 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:
  1. In-order issue with in-order completion.
  2. In-order issue with out-of-order completion.
  3. Out-of-order issue with out-of-order completion.

Policies for Parallel Instruction Execution

- Example: We consider the superscalar architecture:
  - Two instructions can be fetched and decoded at a time;
  - Three functional units can work in parallel: floating point unit, integer adder, integer multiplier;
  - Two instructions can be written back (completed) at a time;
- We consider the following instruction sequence:

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

- 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).

- 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.

19

In-Order Issue with In-Order Completion

- 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.

20

In-Order Issue with In-Order Completion

- 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
```

- Anti-dependency

```
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

- We consider the instruction sequence in above.
- 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).

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

- 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.**
- 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)

25

Final Comments on Superscalars

- The following main techniques are characteristic for superscalar processors:
  1. additional pipelined units which are working in parallel;
  2. out-of-order issue&out-of-order completion;
  3. 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 ~16 instructions are buffered for lookahead.

26

Some Architectures

PowerPC 604

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

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

What is Good and what is Bad with Superscalars ?

Good

- The hardware solves everything:
  - Hardware detects potential parallelism between instructions;
  - Hardware tries to issue as many instructions as possible in parallel.
  - Hardware solves register renaming.
- Binary compatibility
  - If functional units are added in a new version of the architecture or some other improvements have been made to the architecture (without changing the instruction sets), old programs can benefit from the additional potential of parallelism.
  - Why? Because the new hardware will issue the old instruction sequence in a more efficient way.

What is Good and what is Bad with Superscalars ?

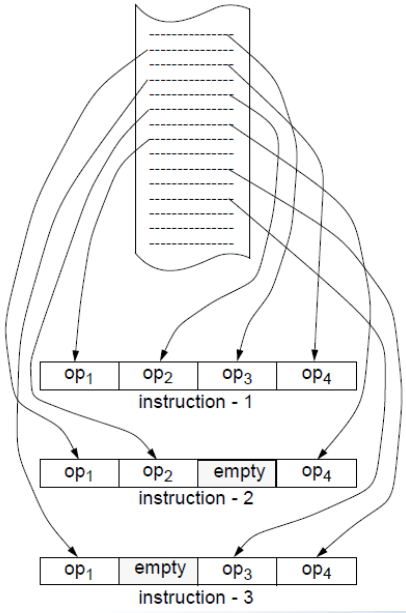
Bad

- Very complex
  - Much hardware is needed for run-time detection. There is a limit in how far we can go with this technique.
  - Power consumption can be very large!
- The window of execution is limited ⇒ this limits the capacity to detect potentially parallel instructions

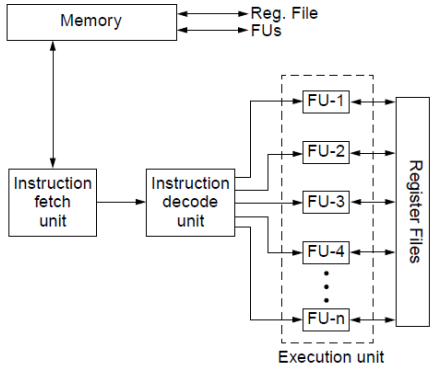
The Alternative: VLIW Processors

- VLIW architectures rely on compile-time detection of parallelism Þ the compiler analysis the program and detects operations to be executed in parallel; such operations are packed into one “large” instruction.
- After one instruction has been fetched all the corresponding operations are issued in parallel.
- No hardware is needed for run-time detection of parallelism.
- The window of execution problem is solved: the compiler can potentially analyse the whole program in order to detect parallel operations.

VLIW Processors



Detection of parallelism and packaging of operations into instructions is done, by the compiler, off-line.

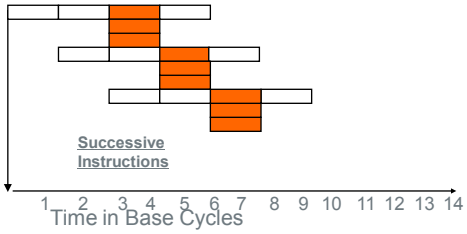


31

Advantages and Problems with VLIW Processors

Advantages

- Simpler hardware:
  - the number of FUs can be increased without needing additional sophisticated hardware to detect parallelism, like in superscalars.
  - Power consumption can be reduced.
- Good compilers can detect parallelism based on global analysis of the whole program (no window of execution problem).



32

Advantages and Problems with VLIW Processors

Problems

- Large number of registers needed in order to keep all FUs active (to store operands and results).
- Large data transport capacity is needed between FUs and the register file and between register files and memory.
- High bandwidth between instruction cache and fetch unit.
  - Example: one instruction with 7 operations, each 24 bits ⇒ 168 bits/instruction.
- Large code size, partially because unused operations ⇒ wasted bits in instruction word.
- Incomputability of binary code
  - For example:
  - If for a new version of the processor additional Fus are introduced ⇒ the number of operations possible to execute in parallel is increased ⇒ the instruction word changes ⇒ old binary code cannot be run on this processor.