CSCI 4717/5717 Computer Architecture

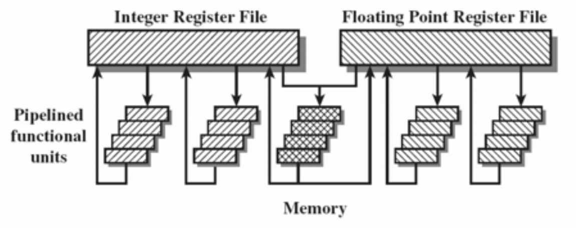
Topic: Instruction Level Parallelism

Reading: Stallings, Chapter 14

What is Superscalar?

- A machine designed to improve the performance of the execution of scalar instructions. (The bulk of instructions.)
- Equally applicable to RISC & CISC, but usually RISC
- Done with multiple pipelines – this is different than multiple pipelines for branching
- Degree = number of pipelines (e.g., degree 2 superscalar pipeline → two pipelines)
- Common instructions (arithmetic, load/store, conditional branch) can be initiated and executed independently

What is Superscalar? (continued)



Why the drive toward Superscalar?

- Most operations are on scalar quantities
- Improving this facet will give us greatest reward

	Pascal	C	Average
Integer constant	16%	23%	20%
Scalar variable	58%	53%	55%
Array/ structure	26%	24%	25%

In class exercise

Develop a short assembly language program (5-6 instructions) where the lines of code could be rearranged or don't depend on one another

What can be done in parallel?

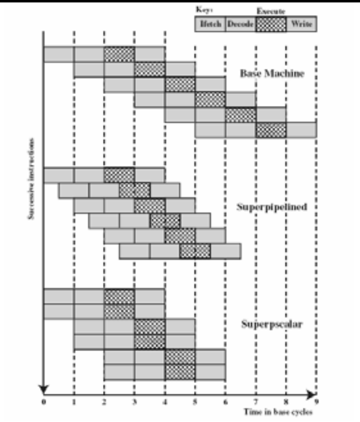
Disregarding the need to use a bus in parallel, what types of instructions are inherently independent?

- Scalar arithmetic and logic with results stored in independent registers
- Transfers of data where bus conflicts are not a problem
- Order of operations can be changed
- Conditional branches

Difference between Superscalar and Super-pipelined

- Many pipeline stages need less than half a clock cycle
- Double internal clock speed gets two tasks per external clock cycle
- Superscalar allows parallel fetch execute

Difference between Superscalar and Super-pipelined (continued)



Instruction level parallelism

- Degree to which instructions of a program can be executed in parallel
- Dependent on
  - Compiler based optimization
  - Hardware techniques

In class exercise

Using the programs you developed a few moments ago, what requirements did you place on the architecture to make the instructions independent?

Limits of Instruction Level Parallelism

Instruction level parallelism is limited by:

- True data dependency
- Procedural dependency
- Resource conflicts
- Output dependency
- Antidependency

True Data Dependency

- True data dependency is where one instruction depends on the final outcome of a previous instruction.
- Also known as flow dependency or write-read dependency
- Consider the code:

```
ADD  r1,r2          (r1 = r1+r2;)
MOV  r3,r1          (r3 = r1;)
```

- Can fetch and decode second instruction in parallel with first
- Can NOT execute second instruction until first is finished

True Data Dependency (continued)

- RISC architecture would reorder following set of instructions or insert delay

```
MOV  r1,[mem]        (Load r1 from memory)
MOV  r3,r1           (r3 = r1;)
MOV  r2,5            (r2 = 5;)
```

- The superscalar machine would execute the first and third instructions in parallel, yet have to wait anyway for the first instruction to finish before executing the second
- This holds up MULTIPLE pipelines

True Data Dependency (continued)

- Is the following an example of data dependency?

```
ADD  r1,r2       (r1 = r1+r2;)
SUB  r3,r1       (r3 = r3-r1;)
```

- Is the following an example of data dependency?

```
ADD  r1,r2       (r1 = r1+r2;)
SUB  r1,r3       (r1 = r1-r3;)
```

- Due to nature of arithmetic, this is more of a resource conflict

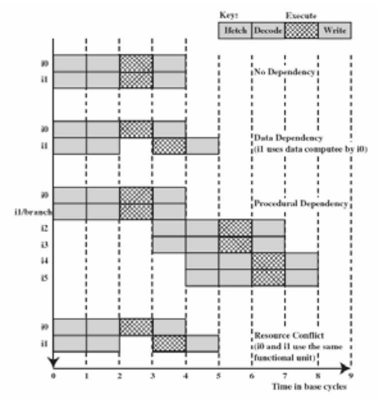
Procedural Dependency

- Can not execute instructions after a branch in parallel with instructions before a branch – this holds up MULTIPLE pipelines
- Variable-length instructions – must partially decode first instruction for first pipe before second instruction for second pipe can be fetched

Resource Conflict

- Two or more instructions requiring access to the same resource at the same time
- Resources include: Memory, Caches, Buses, Registers, Ports, and Functional Units
- Possible solution -- duplicate resources (e.g., two ALUs)

Comparison of True Data, Procedural, and Resource Conflict Dependencies



Output Dependency

- This type of dependency occurs when two instructions both write a result.
- If an instruction depends on the intermediate result, problems could occur
- Also known as write-write dependency
  - R3 = R3 + R5;   (I1)
  - R4 = R3 + 1;   (I2)
  - R3 = R5 + 1;   (I3)
  - R7 = R3 + R4;   (I4)
- I2 depends on result of I1 and I4 depends on result of I3 – true data dependency
- If I3 completes before I1, result from I1 will be written last – output (write-write) dependency

Design Issues

- Instruction level parallelism
  - Instructions in a sequence are independent
  - Execution can be overlapped
  - Governed by data and procedural dependency
- Machine Parallelism
  - Ability to take advantage of instruction level parallelism
  - Governed by number of parallel pipelines ***AND*** by ability to find independent instructions

Instruction Issue Policy

- The protocol used to issue instructions
- Types of orderings include:
  - Order in which instructions are fetched
  - Order in which instructions are executed
  - Order in which instructions change registers and memory
- More sophisticated processor → less bound by relationships of these three orderings
- To optimize pipelines, need to alter one or more of these three with respect to sequential ordering in memory

Instruction Issue Policy (continued)

Three categories of issue 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

In-Order Issue with In-Order Completion

- Issue instructions in the order they occur and write results in same order
- For base-line comparison more than an actual implementation
- Not very efficient – Instructions may stall if:
  - "Partnered" instruction requires more time
  - "Partnered" instruction requires same resource
- Parallelism limited by bottleneck stage (e.g., if can only fetch two instructions at one time, degree of execution parallelism of 3 is never realized)
- This adds to our dependencies issues → Forced order of output

In-Order Issue with In-Order Completion (continued)

Decode		Execute			Write		Cycle
I1	I2						1
I3	I4	I1	I2				2
I3	I4	I1					3
	I4			I3	I1	I2	4
I5	I6			I4			5
	I6		I5		I3	I4	6
			I6				7
					I5	I6	8

In-Order Issue with In-Order Completion (continued)

- Only capable of fetching 2 instructions at a time – Next pair must wait until BOTH of first two are out of fetch pipe
- Execution unit – To guarantee in-order completion, a conflict for resources or a need for multiple cycles stalls issuing of instructions

In-Order Issue with Out-of-Order Completion

- Improve performance in scalar RISC of instructions requiring multiple cycles
- Any number of instructions may be in execution stage at one time → not limited by bottleneck
- Allowing for rearranged outputs creates another dependency → Output dependency
- Output dependency makes instruction issue logic more complex
- Interrupt issue – since instructions are not finished in order, returning after an interrupt may return to instruction where next instruction is already done!

In-Order Issue with Out-of-Order Completion (continued)

Decode		Execute			Write		Cycle
I1	I2						1
I3	I4	I1	I2				2
	I4	I1		I3	I2		3
I5	I6			I4	I1	I3	4
	I6		I5		I4		5
			I6		I5		6
					I6		7

In-Order Issue with Out-of-Order Completion (continued)

- Still only capable of fetching 2 instructions at a time – Next pair must wait until BOTH of first two are out of fetch pipe
- Saved a cycle over in-order issue and in-order completion because ***I3 was not held up waiting for previous instruction pair to complete***
- Instructions no longer stalled for multi-cycle instructions
- This adds to our dependencies issues → Forced order of input

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

- Decouple decode pipeline from execution pipeline with a buffer
- Buffer is called instruction window
- Can continue to fetch and decode until this buffer is full
- When a functional unit becomes available, an instruction is assigned to that pipe to be executed provided:
  - it needs that particular functional unit
  - no conflicts or dependencies are currently blocking its execution
- Since instructions have been decoded, processor can look ahead in hopes of identifying independent instructions.

Out-of-Order Issue with Out-of-Order Completion (continued)

Decode		Window	Execute			Write		Cycle
I1	I2							1
I3	I4	*I1,I2*	I1	I2				2
I5	I6	*I3,I4*	I1		I3	I2		3
		I4,I5,I6		I6	I4	I1	I3	4
		I5		I5		I4	I6	5
						I5		6

Out-of-Order Issue with Out-of-Order Completion (continued)

- Fills fetch pipe as quickly as it can
- I5 depends on output of I4, but I6 is independent and may be executed as soon as functional unit is available. Saves one cycle over in-order issue and out-of-order completion
- Instructions no longer stalled waiting for instruction fetch pipe

Antidependency

- Allowing for rearranged entrance to execution unit → Antidependency (A.K.A. read-write dependency)
- Called Anitdependency because it is the exact opposite of data dependency
- Data dependency: instruction 2 depends on data from instruction 1
- Antidependency: instruction 1 depends on data that could be destroyed by instruction 2

Antidependency (continued)

- Example:

R3 = R3 + R5;	(I1)
R4 = R3 + 1;	(I2)
R3 = R5 + 1;	(I3)
R7 = R3 + R4;	(I4)

- I3 can not complete before I2 starts as I2 needs a value in R3 and I3 changes R3

In class exercise

Identify the write-read, write-write, and read-write dependencies in the instruction sequence below.

```
L1:  R1 ← R2 + R3
L2:  R4 ← R1 + 1
L3:  R1 ← R3 * 2
L4:  R5 ← R1 + R3
L5:  R5 ← R5 + 10
```

"Write" Dependency Problems

Need to solve problems caused by output and anti-dependencies:

- Different than data dependencies which are due to flow of data through a program or sequence of instructions
- Reflect sequence of values in registers which may not reflect the correct ordering from the program
- At any point in an "in-order issue with in-order completion" system, can know what value is in any register at any time
- At any point in system with output and anti-dependencies, cannot know what value is in any register at any time (i.e., program doesn't dictate order of changing data in registers)

Register Renaming

- To fix these problems, processor may need to stall a pipeline stage
- These problems are storage conflicts – multiple instructions competing for use of same register
- Solution – duplicate resources
- Assigning a value to a register dynamically creates new register
- Subsequent reads to that register must go through renaming process

Register Renaming (continued)

- Example

$R3_b = R3_a + R5_a$	(I1)
$R4_b = R3_b + 1$	(I2)
$R3_c = R5_a + 1$	(I3)
$R7_b = R3_c + R4_b$	(I4)

- Without subscript refers to logical register in instruction
- With subscript is hardware register allocated

In class exercise

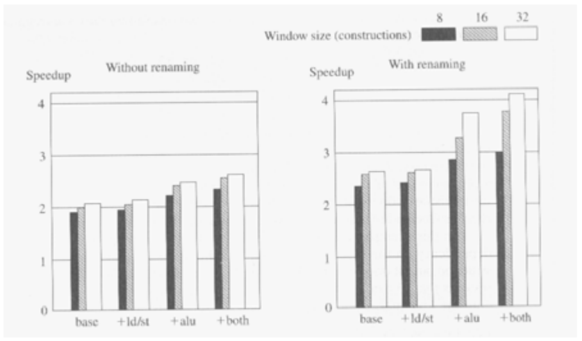
In the code below, identify references to initial register values by adding the subscript 'a' to the register reference. Identify new allocations to registers with the next highest subscript and identify references to these new allocations using the same subscript.

```
R7   = R3   + R4
R3   = R7
R7   = R7   + 1
R4   = R5
R3   = R7   + R3
R5   = R4   + R3
```

Machine Parallelism

- So far, we have discussed three methods for improving performance:
  - duplication of resources
  - out-of-order execution
  - register renaming
- Studies have been conducted to verify the relationships between these methods

Machine Parallelism (continued)



Machine Parallelism (continued)

- Base – No duplicate resources, but can issue instructions out of order
- Not worth duplication functions without register renaming
- Need instruction window large enough (more than 8)

Branch Prediction

Problems with using RISC-type branch delay with superscalar machines

- Branch delay forces pipe always to execute instruction following branch – keeps pipeline full and makes pipeline logic simpler
- Superscalar would have a problem with this as it would execute multiple instructions

Branch Prediction (continued)

Superscalars machines go to pre-RISC techniques of branch prediction

- Prefetch causes two-cycle delay when branch is taken (80486 fetches both next sequential instruction after branch and branch target instruction)
- Older superscalar implementations use static techniques of branch prediction
- More sophisticated processors (PPC 620 and Pentium 4) use dynamic branch prediction based on branch history

Superscalar Implementation

- Simultaneously fetch multiple instructions
  - Branch prediction
  - Pre-decode of instructions for length and branching
  - Multiple fetch mechanism
- Logic to determine true dependencies involving register values – Mechanisms to communicate these values to where they are needed (including register renaming)
- Mechanisms to initiate multiple instructions in parallel
- Resources for parallel execution of multiple instructions
- Mechanisms for committing process state in correct order