CSCI 564 Advanced Computer Architecture

Lecture 10: Out-of-Order Execution

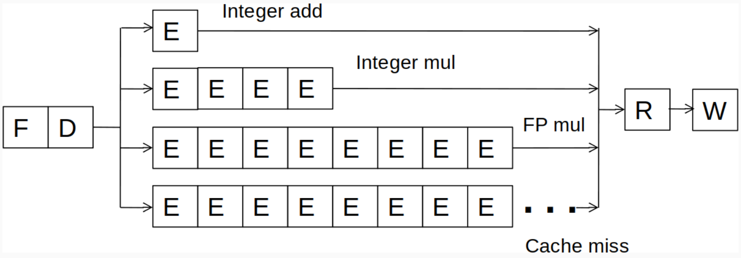
Akshit Sharma (with modifications by Dr. Bo Wu and Ismet Dagli)

March 25, 2024

Colorado School of Mines

Review of In-Order Pipelines

An In-Order Pipeline



- **Problem**: A true data dependency stalls *dispatch* of younger instructions into functional (that is, execution) units.
- **Dispatching** an instruction is the act of sending that instruction to a functional unit.

Know the Enemy I

What do the following two pieces of code have in common (with respect to execution in the previous design)?

```
IMUL R3 <- R1, R2
ADD  R3 <- R3, R1
ADD  R1 <- R6, R7
IMUL R5 <- R6, R8
ADD  R7 <- R9, R9
```

```
LD   R3 <- R1(0)
ADD  R3 <- R3, R1
ADD  R1 <- R6, R7
IMUL R5 <- R6, R8
ADD  R7 <- R9, R9
```

Answer: the first `ADD` stalls the whole pipeline!

- `ADD` cannot dispatch because its source registers are unavailable.
- Later *independent* instructions cannot get executed.

Know the Enemy I

What do the following two pieces of code have in common (with respect to execution in the previous design)?

```
IMUL R3 <- R1, R2
ADD  R3 <- R3, R1
ADD  R1 <- R6, R7
IMUL R5 <- R6, R8
ADD  R7 <- R9, R9
```

```
LD   R3 <- R1(0)
ADD  R3 <- R3, R1
ADD  R1 <- R6, R7
IMUL R5 <- R6, R8
ADD  R7 <- R9, R9
```

Answer: the first `ADD` stalls the whole pipeline!

- `ADD` cannot dispatch because its source registers are unavailable.
- Later *independent* instructions cannot get executed.

Know the Enemy II

What is *different* between the following two pieces of code (with respect to execution in the previous design)?

```
IMUL R3 <- R1, R2
ADD  R3 <- R3, R1
ADD  R1 <- R6, R7
IMUL R5 <- R6, R8
ADD  R7 <- R9, R9
```

```
LD   R3 <- R1(0)
ADD  R3 <- R3, R1
ADD  R1 <- R6, R7
IMUL R5 <- R6, R8
ADD  R7 <- R9, R9
```

Answer: load latency is variable (we don't know if it will be a cache hit or miss until runtime)

- What does this affect? One can be solved by the compiler, the other requires microarchitectural support.

Know the Enemy II

What is *different* between the following two pieces of code (with respect to execution in the previous design)?

```
IMUL R3 <- R1, R2
ADD  R3 <- R3, R1
ADD  R1 <- R6, R7
IMUL R5 <- R6, R8
ADD  R7 <- R9, R9
```

```
LD   R3 <- R1(0)
ADD  R3 <- R3, R1
ADD  R1 <- R6, R7
IMUL R5 <- R6, R8
ADD  R7 <- R9, R9
```

Answer: load latency is variable (we don't know if it will be a cache hit or miss until runtime)

- What does this affect? One can be solved by the compiler, the other requires microarchitectural support.

Preventing Dispatch Stalls

There are many ways to prevent dispatch stalls. We have already seen two:

Preventing Dispatch Stalls

There are many ways to prevent dispatch stalls. We have already seen two:

1. Fine-grained multithreading (execute a different thread each cycle)
2. Compile-time instruction scheduling/reordering

Preventing Dispatch Stalls

There are many ways to prevent dispatch stalls. We have already seen two:

1. Fine-grained multithreading (execute a different thread each cycle)
2. Compile-time instruction scheduling/reordering

What are the disadvantages of the above two?

Preventing Dispatch Stalls

There are many ways to prevent dispatch stalls. We have already seen two:

1. Fine-grained multithreading (execute a different thread each cycle)
2. Compile-time instruction scheduling/reordering

What are the disadvantages of the above two?

- Requires thread saturation
- Not dynamic

Thoughts for how to improve?

Introducing Out-of-Order Execution

Out-of-Order Execution (Dynamic Scheduling)

Idea: Move the dependent instructions out of the way of independent ones.

- Add *rest areas* for dependent instructions (*reservation stations*)

Then, we can monitor the source *values* of each instruction in the rest area.

When all source *values* of an instruction are available, dispatch the instruction.

Benefit: latency tolerance: allows independent instructions to execute and complete in the presence of a long-latency operation.

Out-of-Order Execution (Dynamic Scheduling)

Idea: Move the dependent instructions out of the way of independent ones.

- Add *rest areas* for dependent instructions (*reservation stations*)

Then, we can monitor the source *values* of each instruction in the rest area.

When all source *values* of an instruction are available, dispatch the instruction.

Benefit: latency tolerance: allows independent instructions to execute and complete in the presence of a long-latency operation.

Out-of-Order Execution (Dynamic Scheduling)

Idea: Move the dependent instructions out of the way of independent ones.

- Add *rest areas* for dependent instructions (*reservation stations*)

Then, we can monitor the source *values* of each instruction in the rest area.

When all source *values* of an instruction are available, dispatch the instruction.

Benefit: latency tolerance: allows independent instructions to execute and complete in the presence of a long-latency operation.

In-order vs. Out-of-order Dispatch

- In order dispatch + precise exceptions:

F	D	E	E	E	E	R	W							
	F	D		STALL		E	R	W						
		F		STALL		D	E	R	W					
						F	D	E	E	E	E	R	W	
							F	D		STALL		E	R	W

IMUL R3 ← R1, R2
ADD R3 ← R3, R1
ADD R1 ← R6, R7
IMUL R5 ← R6, R8
ADD R7 ← R3, R5

- Out-of-order dispatch + precise exceptions:

F	D	E	E	E	E	R	W				
	F	D		WAIT		E	R	W			
		F	D	E	R				W		
			F	D	E	E	E	E	R	W	
				F	D		WAIT		E	R	W

- 15 vs. 12 cycles

Data Dependence

Flow dependence

$$\begin{tikzcd}
r_3 & \leftarrow & r_1 & \mathrm{op} & r_2 \\
r_5 & \leftarrow & r_3 \arrow[ull] & \mathrm{op} & r_4
\end{tikzcd}$$

Read-after-Write (RAW)

Anti dependence

$$\begin{tikzcd}
r_3 & \leftarrow & r_1 & \mathrm{op} & r_2 \\
r_1 \arrow[urr] & \leftarrow & r_4 & \mathrm{op} & r_5
\end{tikzcd}$$

Write-after-Read (WAR)

Output-dependence

$$\begin{tikzcd}
r_3 & \leftarrow & r_1 & \mathrm{op} & r_2 \\
r_5 & \leftarrow & r_3 & \mathrm{op} & r_4 \\
r_3 \arrow[uu, bend left] & \leftarrow & r_6 & \mathrm{op} & r_7
\end{tikzcd}$$

Write-after-Write (WAW)

Register Renaming

```
ADD R1 <- R2, R3
ADD R1 <- R4, R5
```

```
ADD R1 <- R2, R3
ADD R6 <- R4, R5
```

Register Renaming

```
ADD R1 <- R2, R3
ADD R1 <- R4, R5
ADD R8 <- R1, R5
```

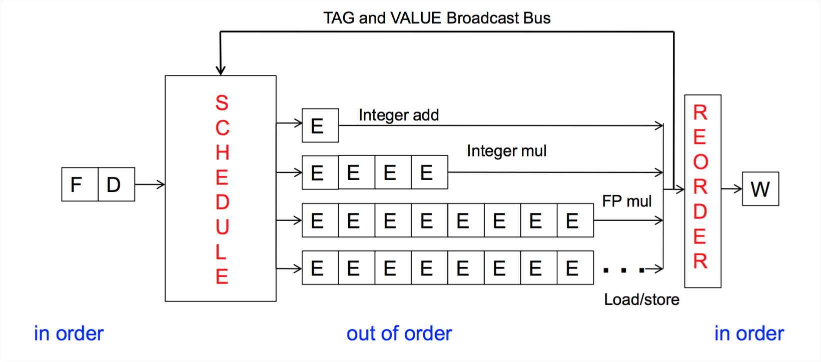
```
ADD R1 <- R2, R3
ADD R6 <- R4, R5
```

Register Renaming

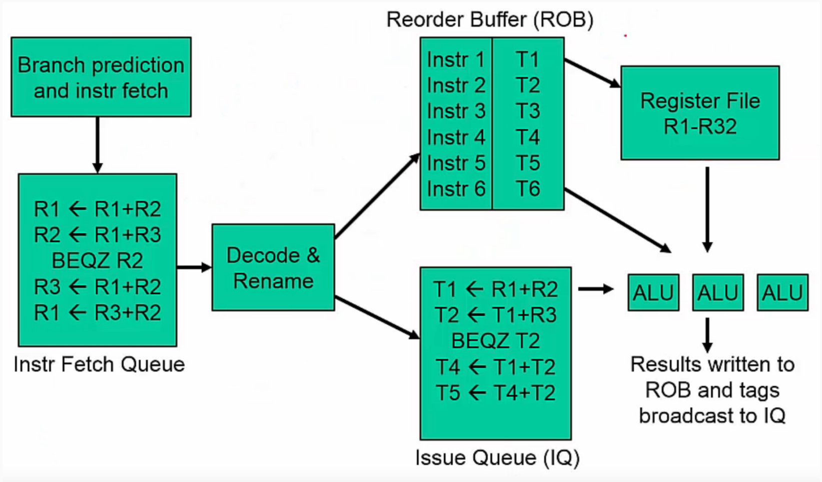
```
ADD R1 <- R2, R3
ADD R1 <- R4, R5
ADD R8 <- R1, R5
```

```
ADD R1 <- R2, R3
ADD R6 <- R4, R5
ADD R8 <- R6, R5
```

Overview of Out-of-Order Execution



Out-of-Order Implementation



Practice: Rename Registers: Worksheet Problem 1

Show the renamed versions of the following code. Assume you have 4 rename registers, T1-T4.

```
ADD   R1, R2, R3
ADD   R3, R4, R5
BEQZ  R1, 1000
ADD   R1, R1, R3
ADD   R1, R1, R3
ADD   R3, R1, R3
```

Practice: Rename Registers: Worksheet Problem 2

Show the renamed versions of the following code. Assume you have 36 physical registers and 32 architected registers.

```
ADD   R1, R2, R3
ADD   R3, R4, R5
BEQZ  R1, 1000
ADD   R1, R1, R3
ADD   R1, R1, R3
ADD   R3, R1, R3
ADD   R4, R3, R1
```

The Dataflow Model

The Dataflow Model

- Von Neumann model: An instruction is fetched and executed in control flow order
  - As specified by the instruction pointer
  - Sequential unless explicit control flow instruction

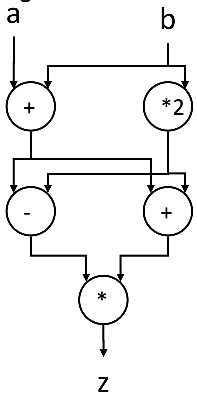
- Dataflow model: An instruction is fetched and executed in data flow order
  - i.e., when its operands are ready
  - i.e., there is no instruction pointer
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies "who" should receive the result
    - An instruction can "fire" whenever all operands are received
  - Potentially many instructions can execute at the same time
    - Inherently more parallel

von Neumann vs Dataflow

- Consider a von Neumann program
  - What is the significance of the program order?
  - What is the significance of the storage locations?

```
v <= a + b;
w <= b * 2;
x <= v - w
y <= v + w
z <= x * y
```

Sequential



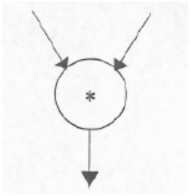
Dataflow

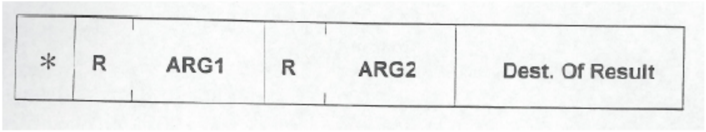
- Which model is more natural to you as a programmer?

More on Data Flow

- In a data flow machine, a program consists of data flow nodes
  - A data flow node fires (fetched and executed) when all its inputs are ready
    - i.e. when all inputs have tokens

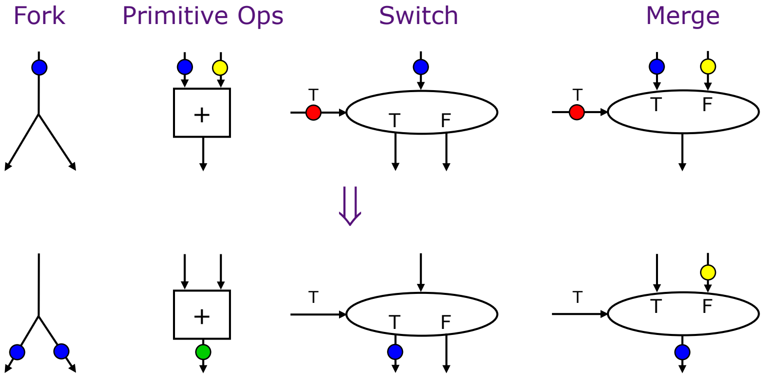
- Data flow node and its ISA representation





Dataflow Nodes

- A small set of dataflow operators can be used to define a general programming language



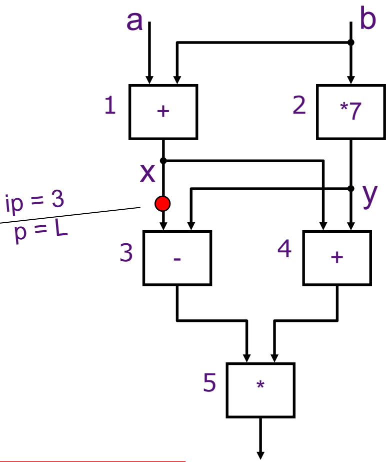
Dataflow Graphs

```
{x = a + b;
 y = b * 7
in
   (x-y) * (x+y)}
```

- Values in dataflow graphs are represented as tokens

token < ip , p , v >

instruction ptr — port — data



- An operator executes when all its input tokens are present; copies of the result token are distributed to the destination operators

no separate control flow

Control Flow vs. Data Flow

$$
\begin{aligned}
a &:= x + y \\
b &:= a \times a \\
c &:= 4 - a
\end{aligned}
$$

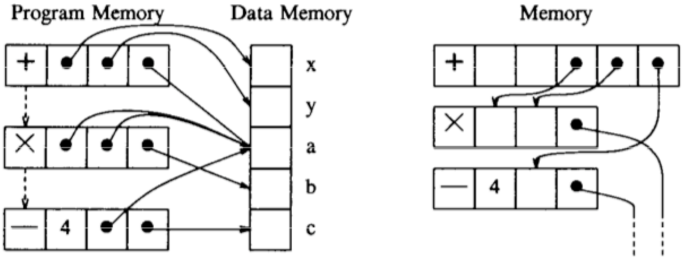
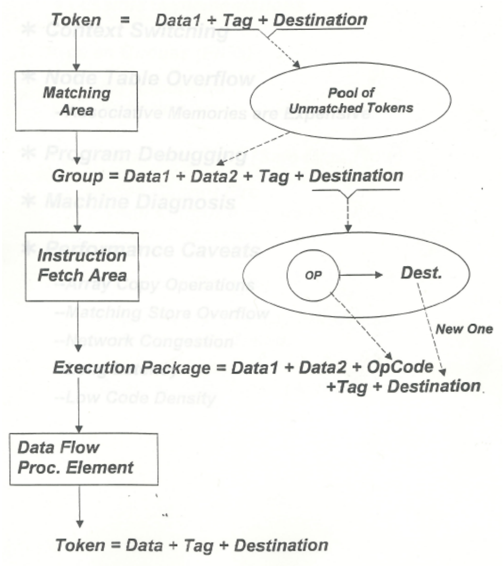


Figure 2. A comparison of control flow and dataflow programs. On the left a control flow program for a computer with memory-to-memory instructions. The arcs point to the locations of data that are to be used or created. Control flow arcs are indicated with dashed arrows; usually most of them are implicit. In the equivalent dataflow program on the right only one memory is involved. Each instruction contains pointers to all instructions that consume its results.

Data Flow Characteristics

- Data-driven execution of instruction-level graphical code
  - Nodes are operators
  - Arcs are data (I/O)
  - As opposed to control-driven execution
- Only real dependencies constrain processing
- No sequential instruction stream
  - No program counter
- Execution triggered by the presence/readiness of data

A Dataflow Processor


Token = Data1 + Tag + Destination
Matching Area
Pool of Unmatched Tokens
Group = Data1 + Data2 + Tag + Destination
Instruction Fetch Area
OP
Dest.
New One
Execution Package = Data1 + Data2 + OpCode +Tag + Destination
Data Flow Proc. Element
Token = Data + Tag + Destination

Data Flow Advantages/Disadvantages

- Advantages
  - Very good at exploiting irregular parallelism
  - Only real dependencies constrain processing

- Disadvantages
  - No precise state
    - Debugging very difficult
    - Interrupt/exception handling is difficult
  - Bookkeeping overhead (tag matching)
  - Too much parallelism? (Parallelism control needed)