Instruction-level parallelism: Introduction

Dr. Tao Xie

Fall, 2017

These slides are adapted from notes by Dr. David Patterson (UCB)

Ideas To Reduce Stalls

Pipeline CPI = Ideal pipeline CPI + Structure stalls + Data hazard stalls + Control stalls

Technique	**Reduces**
Dynamic scheduling	Data hazard stalls
Dynamic branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI
Speculation	Data and control stalls
Dynamic memory disambiguation	Data hazard stalls involving memory
Loop unrolling	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Compiler dependence analysis	Ideal CPI and data hazard stalls
Software pipelining and trace scheduling	Ideal CPI and data hazard stalls
Compiler speculation	Ideal CPI, data and control stalls

Forms of Parallelism

- Process-level
- Thread-level
- Loop-level
- Instruction-level
  - Focus of Chapter 2 & 3

Coarse grain

Human intervention?

Fine Grain

Instruction Level Parallelism (ILP)

Principle: *There are many instructions in code that don't depend on each other*. That means it's possible to execute those instructions in parallel.

- **Instruction-Level Parallelism (ILP): overlap the execution of instructions to improve performance**

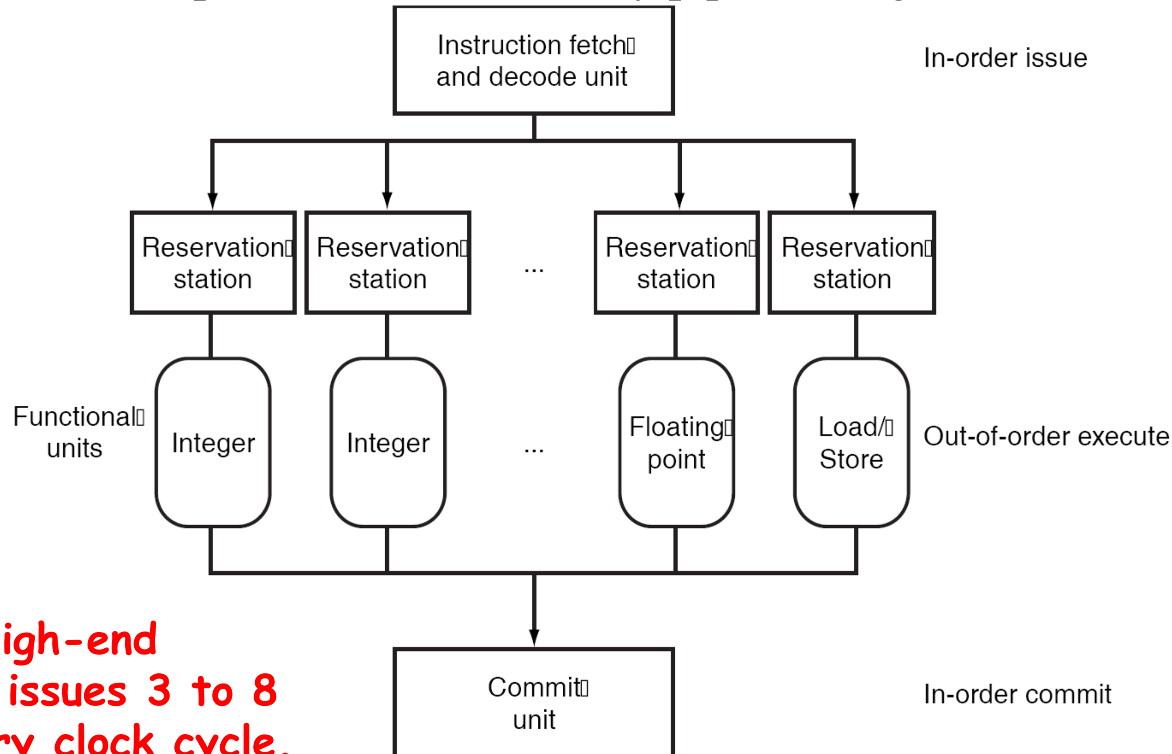
This is easier said than done. Issues include:

- Building compilers to analyze the code,
- Building hardware to be even smarter than that code.

This section looks at some of the problems to be solved.

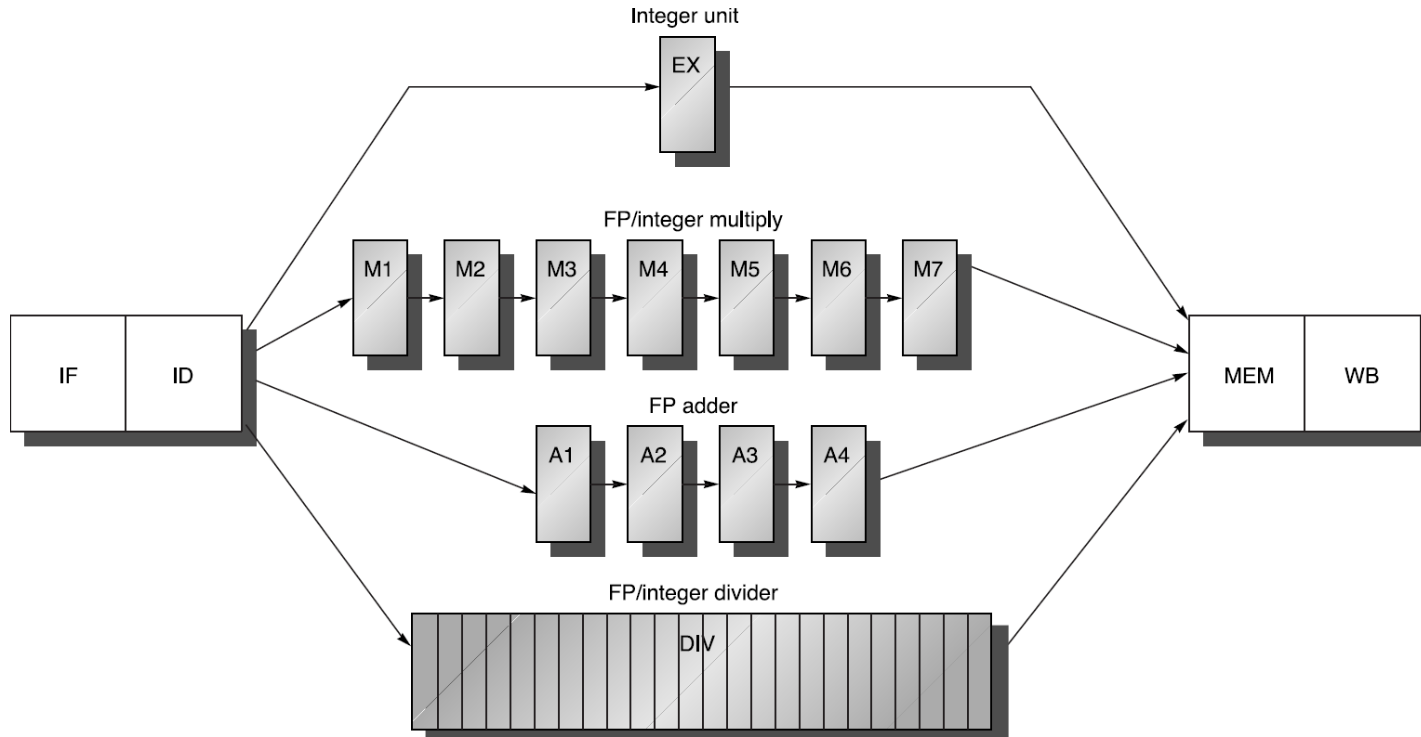
Exploiting Parallelism in Pipeline

- Two methods of exploiting the parallelism ?
  - –Increase pipeline depth
  - –Multiple issue
    - •Replicate internal components
    - •launch multiple instructions in every pipeline stage



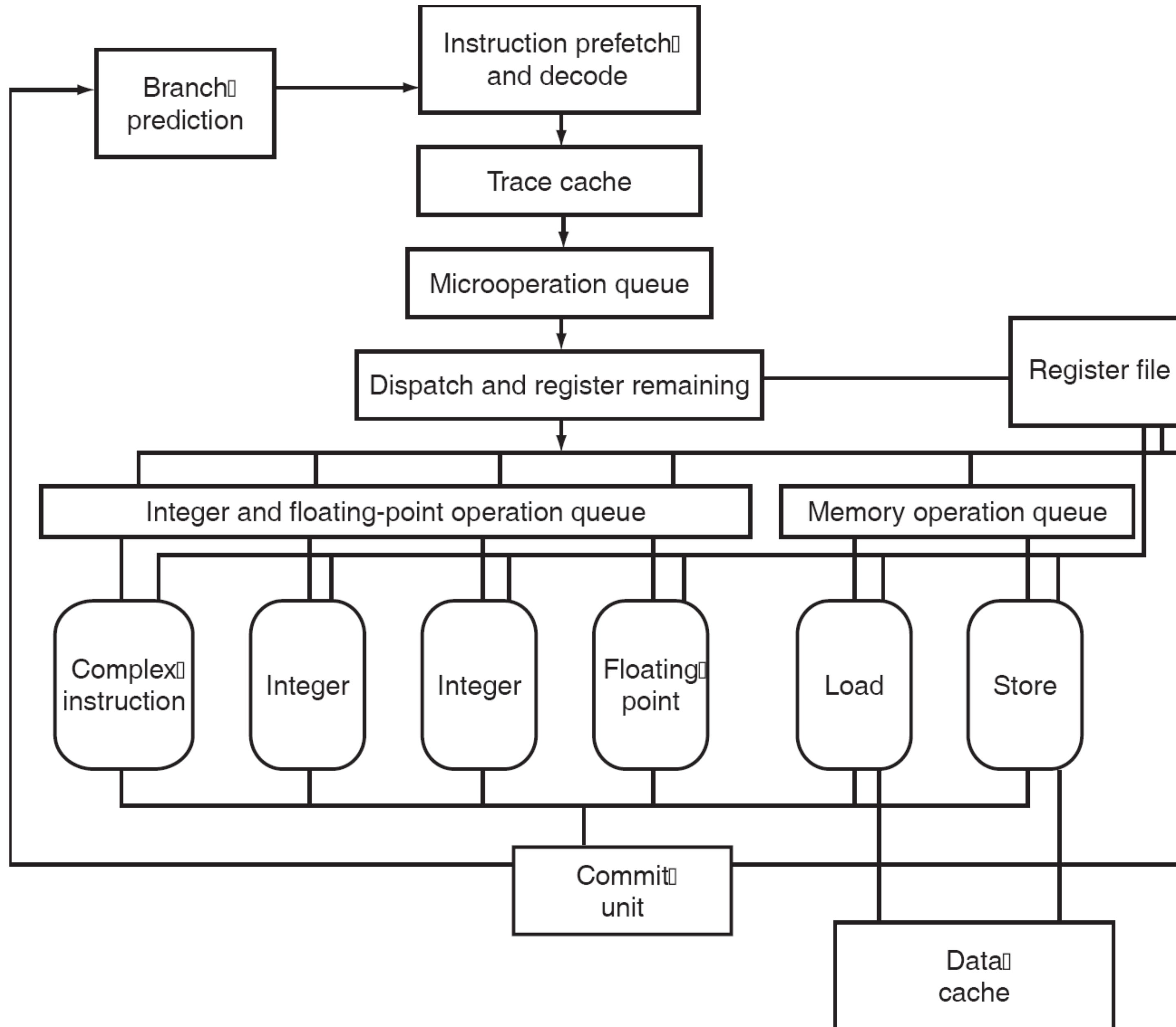
Today's high-end microprocessor issues 3 to 8 instructions every clock cycle.

Pipeline supports multiple outstanding FP operations

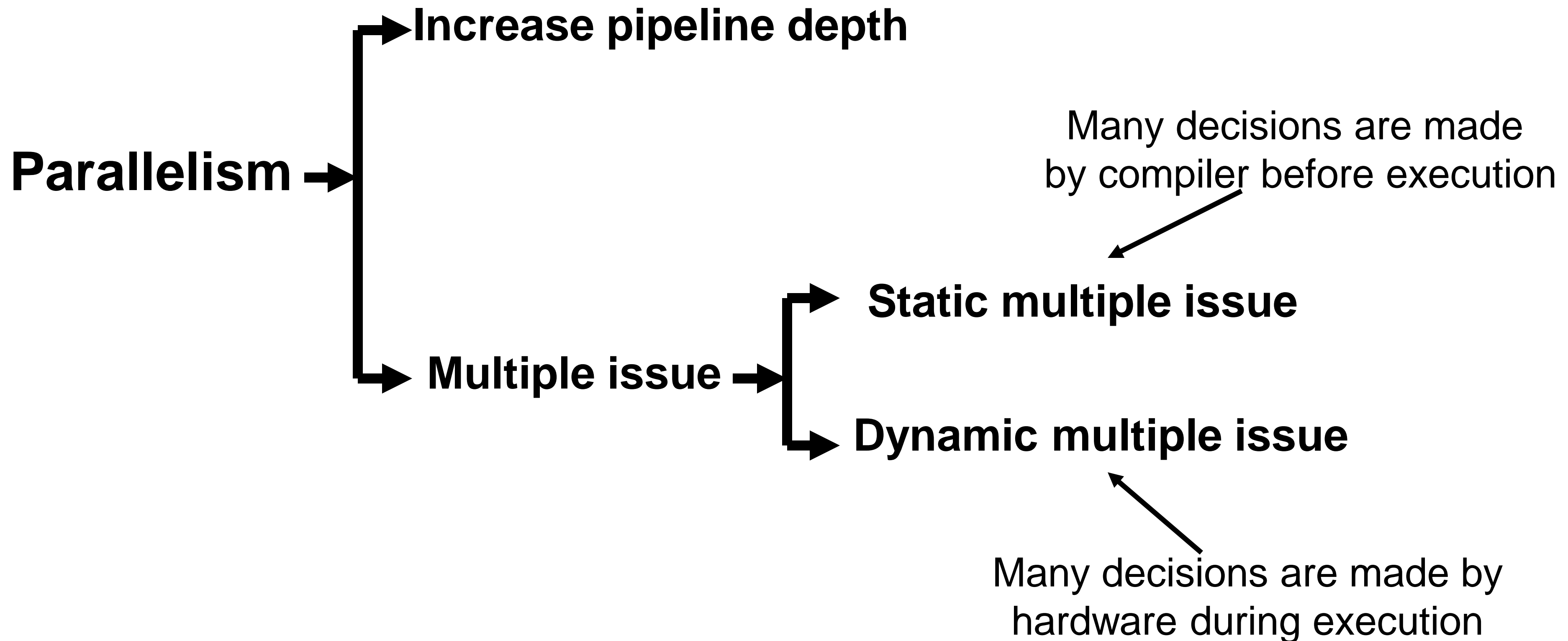


MULTD	IF	ID	M1	M2	M3	M4	M5	M6	M7	Mem	WB
ADDD		IF	ID	A1	A2	A3	A4	Mem	WB		
LD			IF	ID	EX	Mem	WB				
SD				IF	ID	EX	Mem	WB			

Microarchitecture of Intel Pentium 4



The Big Picture

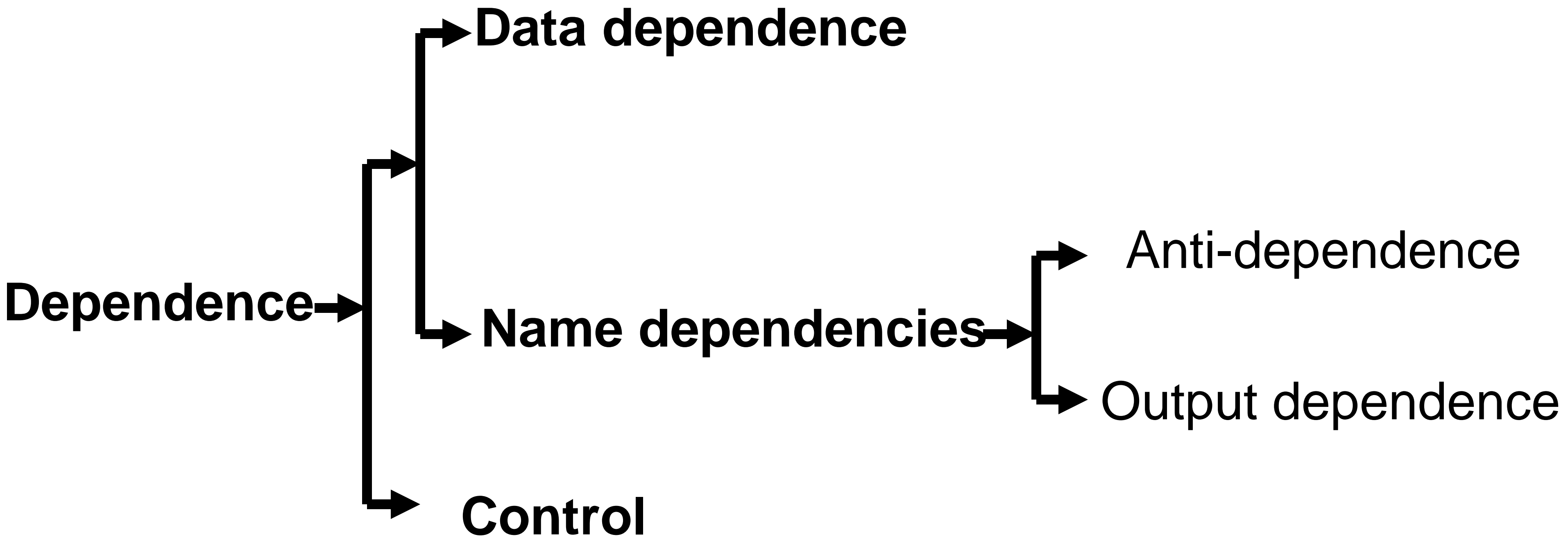


ILP Challenges

- How many instructions can we execute in parallel?
- Definition of Basic instruction block: What is between two branch instructions:
  - Example: Body of a loop.
  - Typical MIPS programs have 15-25 % branch instruction :
    - One every 4-7 instructions is a branch.
    - How many of those are likely to be data dependent on each other?
  - We need the means to exploit parallelism across basic blocks. What stops us from doing so?

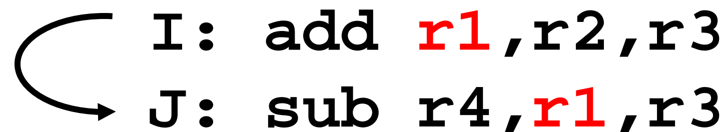
Dependencies

Dependencies



Data Dependence and Hazards

- $Instr_J$ is data dependent on $Instr_I$
  $Instr_J$ tries to read operand before $Instr_I$ writes it



- or $Instr_J$ is data dependent on $Instr_K$ which is dependent on $Instr_I$
- Caused by a "True Dependence" (compiler term)
- If true dependence caused a hazard in the pipeline, called a Read After Write (RAW) hazard

How to detect a True Dependence?

Data Dependences through registers/memory

- Dependences through registers are easy:

  lw r10,10(r11)

  add r12,r10,r8

  just compare register names.

- Dependences through memory are harder:

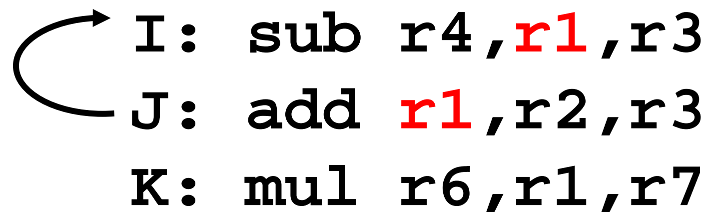
  sw r10,4 (r2)

  lw r6,0(r4)

  is r2+4 = r4+0? If so they are dependent, if not, they are not.

Name Dependence #1: Anti-dependence

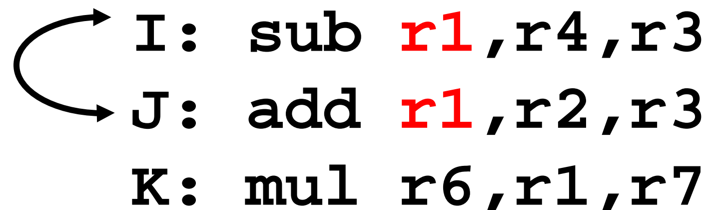
- Name dependence: when 2 instructions use same register or memory location, called a name, but no flow of data between the instructions associated with that name; 2 versions of name dependence
- Instr_J writes operand *before* Instr_I reads it



- Called an "anti-dependence" by compiler writers.
  This results from reuse of the name "r1"
- If anti-dependence caused a hazard in the pipeline, called a Write After Read (WAR) hazard

Name Dependence #2: Output dependence

- $Instr_J$ writes operand ***before*** $Instr_I$ writes it.



```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1"
- If anti-dependence caused a hazard in the pipeline, called a Write After Write (WAW) hazard

Dependences and hazards

- Dependences are a property of programs.
- If two instructions are data dependent they cannot execute simultaneously.
- Whether a dependence results in a hazard and whether that hazard actually causes a stall are properties of the pipeline organization.
- Data dependences may occur through registers or memory.

Dependences and hazards

- The presence of the dependence indicates the potential for a hazard, but the actual hazard and the length of any stall is a property of the pipeline. A data dependence:
  - Indicates that there is a possibility of a hazard.
  - Determines the order in which results must be calculated, and
  - Sets an upper bound on the amount of parallelism that can be exploited.

Instruction Dependence Example

- For the following code identify all data and name dependence between instructions and give the dependency graph

```
1    L.D      F0, 0 (R1)
2    ADD.D    F4, F0, F2
3    S.D      F4, 0(R1)
4    L.D      F0, -8(R1)
5    ADD.D    F4, F0, F2
6    S.D      F4, -8(R1)
```

Please find dependencies!

Instruction Dependence Example

- For the following code identify all data and name dependence between instructions and give the dependency graph

```
1   L.D     F0, 0 (R1)
2   ADD.D   F4, F0, F2
3   S.D     F4, 0(R1)
4   L.D     F0, -8(R1)
5   ADD.D   F4, F0, F2
6   S.D     F4, -8(R1)
```

True Data Dependence:

Instruction 2 depends on instruction 1 (instruction 1 result in F0 used by instruction 2), Similarly, instructions (4,5)
Instruction 3 depends on instruction 2 (instruction 2 result in F4 used by instruction 3), Similarly, instructions (5,6)

Name Dependence:

Output Name Dependence (WAW):

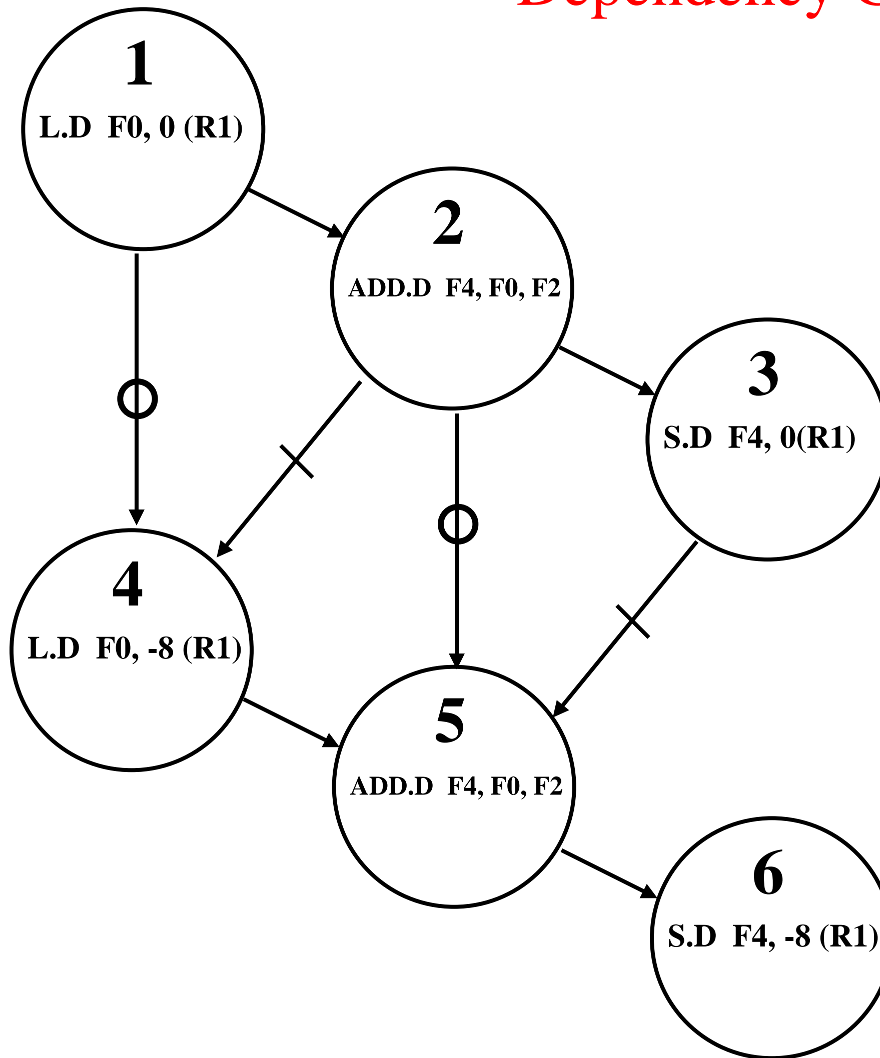
Instruction 1 has an output name dependence over result register (name) F0 with instructions 4
Instruction 2 has an output name dependence over result register (name) F4 with instructions 5

Anti-dependence (WAR):

Instruction 2 has an anti-dependence with instruction 4 over register (name) F0 which is an operand of instruction 1 and the result of instruction 4
Instruction 3 has an anti-dependence with instruction 5 over register (name) F4 which is an operand of instruction 3 and the result of instruction 5

Instruction Dependence Example

Dependency Graph



Example Code

1	L.D	F0, 0 (R1)
2	ADD.D	F4, F0, F2
3	S.D	F4, 0(R1)
4	L.D	F0, -8(R1)
5	ADD.D	F4, F0, F2
6	S.D	F4, -8(R1)

Date Dependence:
(1, 2) (2, 3) (4, 5) (5, 6)

Output Dependence:
(1, 4) (2, 5)

Anti-dependence:
(2, 4) (3, 5)

Three Questions for Last Slide

1. Can instruction 4 (second L.D) be moved just after instruction 1 (first L.D)? If not, what dependencies are violated?
2. Can instruction 3 (first S.D) be moved just after instruction 4 (second L.D)?
3. How about moving 3 after 5 (the second ADD.D)? If not what dependencies are violated?

No, instruction 4 cannot be moved just after instruction 1 because this changes the original value of F0. Output dependence is violated.

Yes, instruction 3 can be moved just after instruction 4 because 3 and 4 reference to different memory locations.

No, instruction 3 cannot be moved after 5 because 5 changes the content of F4. Anti-dependence is violated.

ILP and Data Hazards

- HW/SW must preserve program order:
  order instructions would execute as if executed sequentially 1 at a time as determined by original source program
- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program
- Instructions involved in a name dependence can execute simultaneously (how to implement?)

  - if name used in instructions is changed so instructions do not conflict
    - Register renaming resolves name dependence for registers
    - Either by compiler or by HW

Control Dependencies

- Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order

```
if p1 {
   S1;
};
if p2 {
   S2;
}
```

- S1 is control dependent on p1, and
- S2 is control dependent on p2 but not on p1.

Control Dependence Ignored

- Control dependence need not be preserved
  - willing to execute instructions that should not have been executed, thereby violating the control dependences, if can do so without affecting correctness of the program

```
  DADDU r2,r3,r4
  beqz r2,l1
  lw r1,0(r2)
l1:
```

Can we move lw before the branch?

Don't worry, it is OK to violate control dependences as long as we can preserve the program semantics

Instead, 2 properties critical to program correctness are exception behavior and data flow

Preserving the exception behavior

- Corollary:

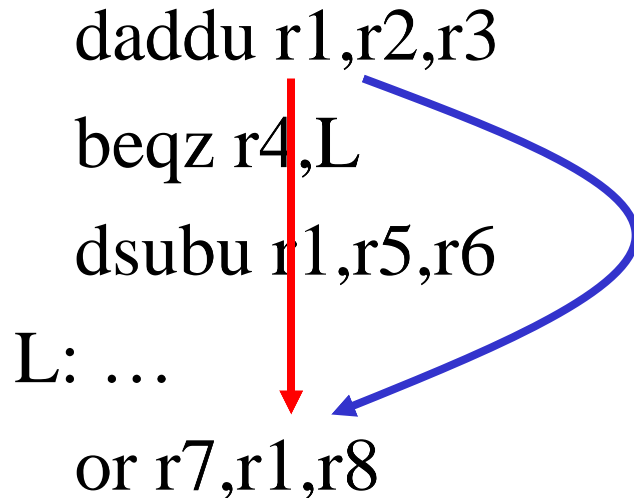
  Any changes in the ordering of instructions should not change how exceptions are raised in a program.

```
    DADDU r2,r3,r4
    beqz r2,l1
    lw r1,0(r2)
l1:
```

→ Reordering of instruction execution should not cause any new exceptions.

Preserving the data flow

- Consider the following example:



- What can you say about the value of r1 used by the or instruction?

Preserving the data flow

- Corollary:

  Preserving data dependences alone is not sufficient when changing program order. We must preserve the data flow.

- Data flow: actual flow of data values among instructions that produce results and those that consume them.
- These two principles together allow us to execute instructions in a different order and still maintain the program semantics.
- This is the foundation upon which ILP processors are built.

Speculation

```
          DADDU R1, R2, R3
          BEQZ R12, skipnext
          DSUBU R4, R5, R6
          DADDU R5, R4, R9
skipnext  OR R7, R8, R9
```

- Assume R4 is *dead* (rather than live) after skipnext.
- We can execute DSUBU before BEQZ since
  - R4 could not generate an exception.
  - The data flow cannot be affected.
- This type of code scheduling is called *speculation.*
  - The compiler is betting on the branch outcome. In this case, the bet is that the branch is usually not take.

Summary

- Two critical properties to maintain program correctness:

  1. Any changes in the ordering of instructions should not change how exceptions are raised in a program.

  2. The data flow is preserved.

Exercise

Identify each dependence by type; list the two instructions involved; identify which instruction is dependent; and, if there is one, name the storage location involved.

```
LD      R1, 45(R2)
DADD    R7, R1, R5
DSUB    R8, R1, R6
OR      R9, R5, R1
BNEZ    R7, target
DADD    R10, R8, R5
XOR     R2, R3, R4
```

	Dependence type	Independent instruction	Dependent instruction	Storage location
1	data	LD	DADD	R1

There are totally 8 dependences!