# CS510 Computer Architecture

# Lecture 07: Pipelining III & ILP

**Soontae Kim**

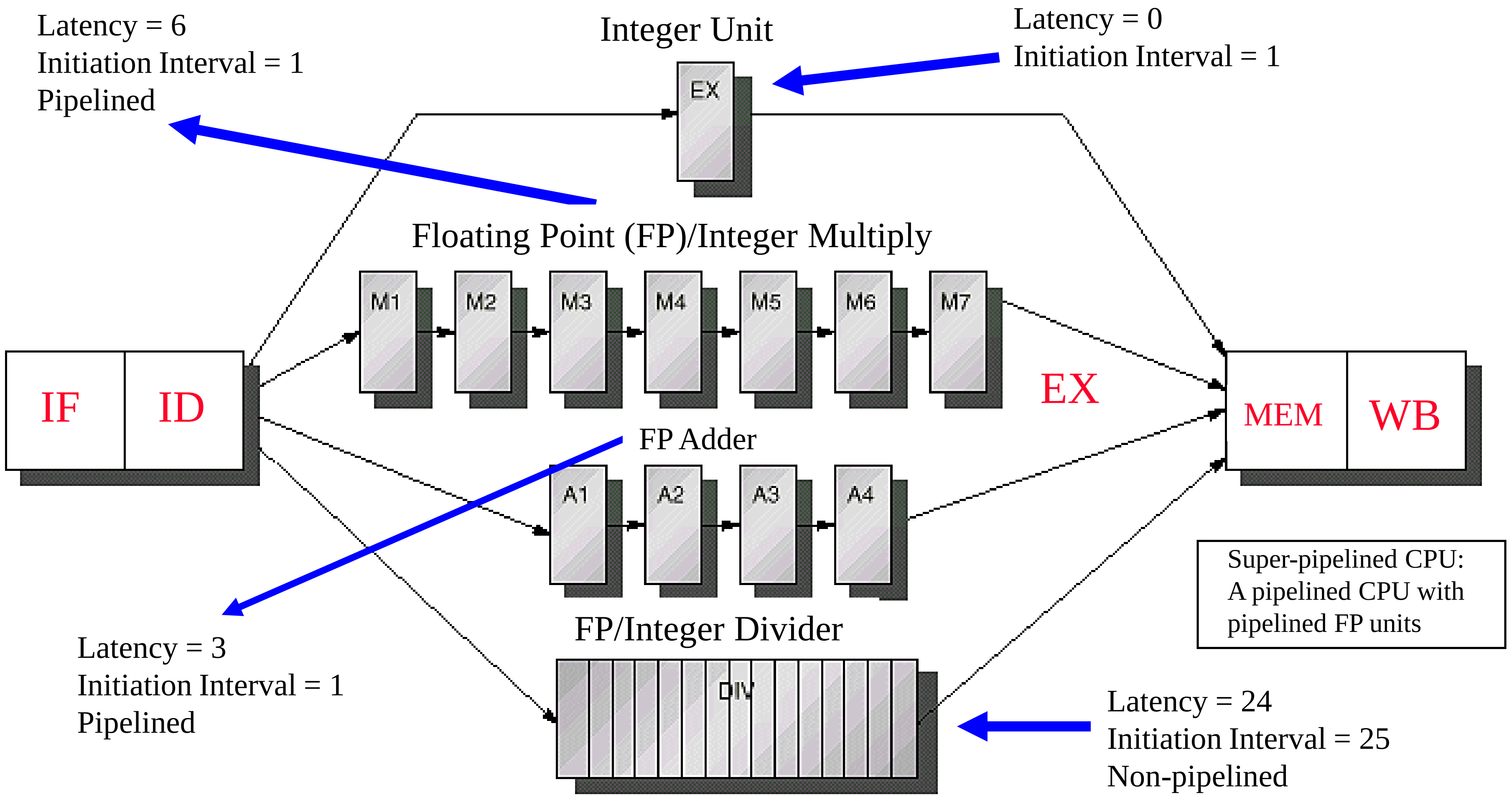
**Spring 2017**

**School of Computing, KAIST**

## Notice

- **Homework assignment#1**
  - Due on March 31 (Friday)
  - Available on class web site

# Extending The MIPS Pipeline: Multiple Outstanding Floating Point Operations



Super-pipelined CPU:
A pipelined CPU with
pipelined FP units

**A pipeline that supports multiple outstanding FP operations.**

In-Order Single-Issue MIPS Pipeline with FP Support

# Pipeline Characteristics With FP Support

- **Instructions are still processed in-order in IF, ID, EX at the rate of one instruction per cycle.**
- **<u>Longer RAW hazard stalls</u> due to long FP latencies.**
- **<u>Structural hazards possible</u> due to varying instruction and FP latencies:**
  - **FP unit may not be available; divide in this case.**
  - **MEM, WB reached by several instructions simultaneously.**
- **<u>WAW hazards can occur</u> since it is possible for instructions to reach WB out-of-order.**
- **<u>WAR hazards impossible</u>, since register reads occur in-order in ID.**
- **Instructions can be allowed to <u>complete out-of-order</u>.**

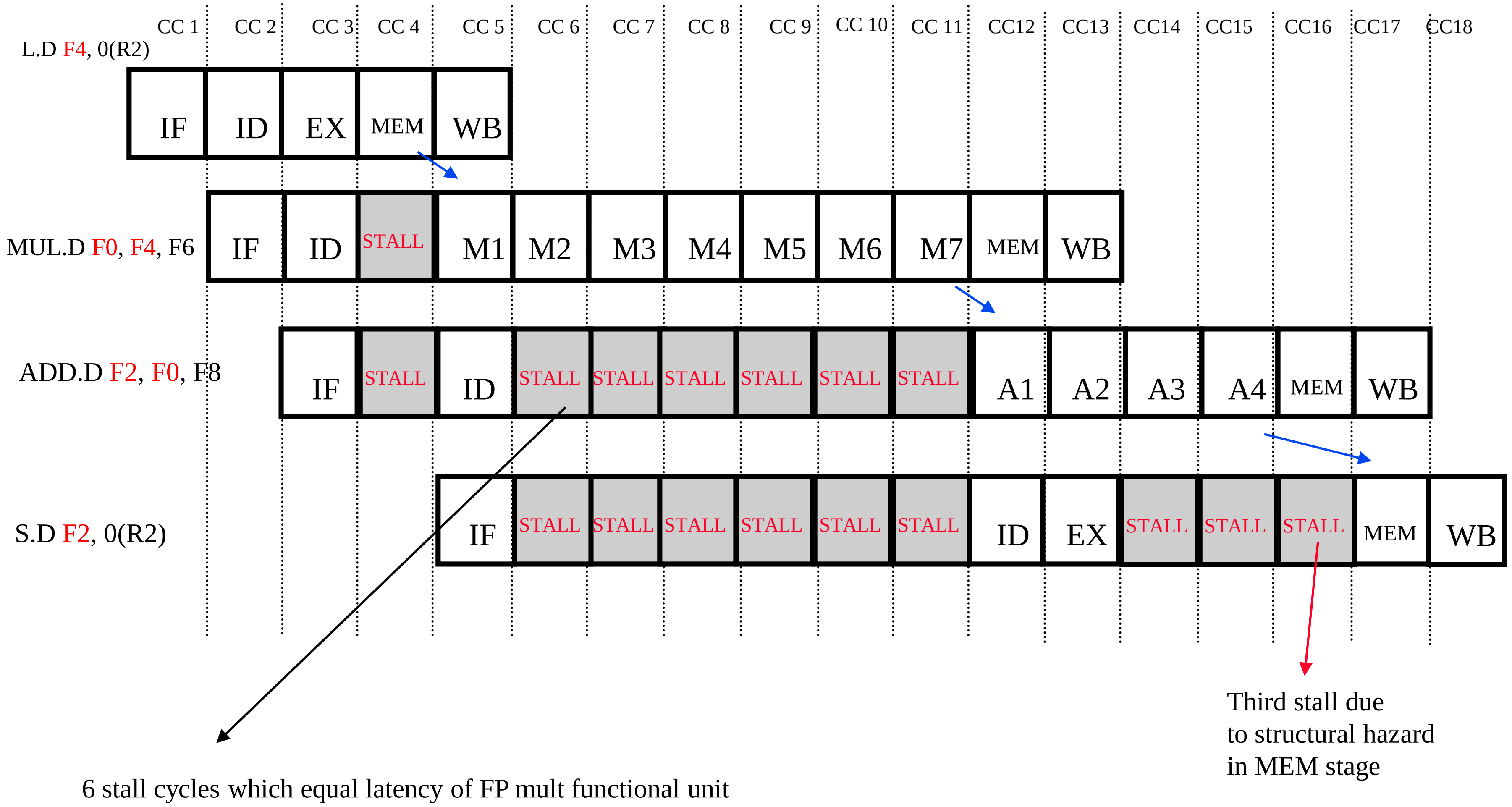
# FP Operations Pipeline Timing Example

| | CC 1 | CC 2 | CC 3 | CC 4 | CC 5 | CC 6 | CC 7 | CC 8 | CC 9 | CC 10 | CC 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MUL.D | IF | ID | M1 | M2 | M3 | M4 | M5 | M6 | M7 | MEM | WB |
| ADD.D | | IF | ID | A1 | A2 | A3 | A4 | MEM | WB | | |
| L.D | | | IF | ID | EX | MEM | WB | | | | |
| S.D | | | | IF | ID | EX | MEM | WB | | | |

All above instructions are assumed independent out of order program

# FP Code RAW Hazard Stalls Example

**(with full data forwarding in place)**



| | CC 1 | CC 2 | CC 3 | CC 4 | CC 5 | CC 6 | CC 7 | CC 8 | CC 9 | CC 10 | CC 11 | CC12 | CC13 | CC14 | CC15 | CC16 | CC17 | CC18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| L.D F4, 0(R2) | IF | ID | EX | MEM | WB | | | | | | | | | | | | | |
| MUL.D F0, F4, F6 | | IF | ID | STALL | M1 | M2 | M3 | M4 | M5 | M6 | M7 | MEM | WB | | | | | |
| ADD.D F2, F0, F8 | | | IF | STALL | ID | STALL | STALL | STALL | STALL | STALL | STALL | A1 | A2 | A3 | A4 | MEM | WB | |
| S.D F2, 0(R2) | | | | | IF | STALL | STALL | STALL | STALL | STALL | STALL | ID | EX | STALL | STALL | STALL | MEM | WB |

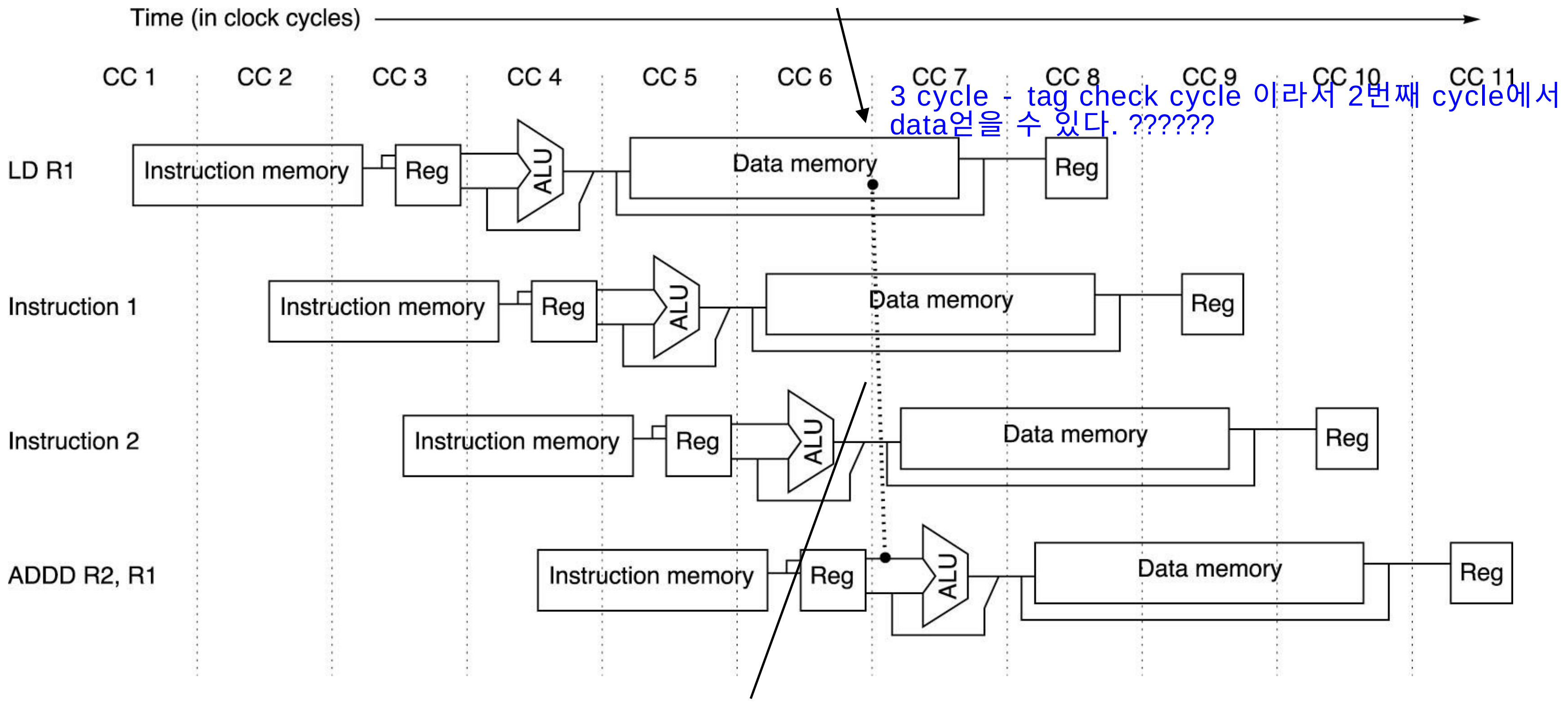
6 stall cycles which equal latency of FP mult functional unit
Third stall due to structural hazard in MEM stage

# FP Code Structural Hazards Example

| | CC 1 | CC 2 | CC 3 | CC 4 | CC 5 | CC 6 | CC 7 | CC 8 | CC 9 | CC 10 | CC 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MULT.D F0, F4, F6 | IF | ID | M1 | M2 | M3 | M4 | M5 | M6 | M7 | MEM | WB |
| . . . (integer) | | IF | ID | EX | MEM | WB | | | | | |
| . . . (integer) | | | IF | ID | EX | MEM | WB | | | | |
| ADD.D F2, F4, F6 | | | | IF | ID | A1 | A2 | A3 | A4 | MEM | WB |
| . . . (integer) | | | | | IF | ID | EX | MEM | WB | | |
| . . . (integer) | | | | | | IF | ID | EX | MEM | WB | |
| LD F2, 0(R2) | | | | | | | IF | ID | EX | MEM | WB |

# MIPS R4000 Example

LW data available here



Forwarding of LW Data

- **Even with forwarding the deeper pipeline leads to a 2-cycle load-use delay (2 stall cycles).**

# Pipelining and Exploiting Instruction-Level Parallelism (ILP)

- Instruction-Level Parallelism (ILP) exists when instructions in a sequence are independent and thus can be executed in parallel.
  - Pipelining increases performance by overlapping the execution of independent instructions and thus exploits ILP in the code.
- Preventing instruction dependence violations (hazards) may result in stall cycles in a pipelined CPU increasing its CPI.
  - The CPI of a real-life pipeline is given by (assuming ideal memory):

$$\boxed{\begin{aligned}\text{Pipeline CPI} = {} & \text{Ideal Pipeline CPI} + \text{Structural Stalls} + \text{RAW Stalls} \\ & + \text{WAR Stalls} + \text{WAW Stalls} + \text{Control Stalls}\end{aligned}}$$

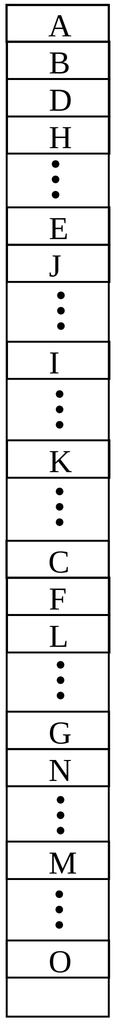
- Programs that have more ILP tend to perform better on pipelined CPUs.
  - More ILP mean fewer instruction dependencies and thus fewer stall cycles needed to prevent instruction dependence violations

# Basic Instruction Block

- ***A basic instruction block*** **is a straight-line code sequence with no branches going-in except to the entry and with no branches going-out except at the exit point.**
  - **Example: Body of a loop.**
- **The amount of instruction-level parallelism (ILP) in a basic block is limited by instruction dependence present and size of the basic block.**
- **In typical integer code, dynamic branch frequency is between 15% and 25% (resulting in average basic block size of 4 to 7 instructions).** branch 25% = average basic block size 4 => inst1 inst2 inst3 branch
- **Any static technique that increases the average size of basic blocks, which increases the amount of ILP in the code, provides more instructions for static pipeline scheduling by the compiler, possibly eliminates more stall cycles, and thus improves pipelined CPU performance.**
  - **Loop unrolling is one such technique that we examine next**

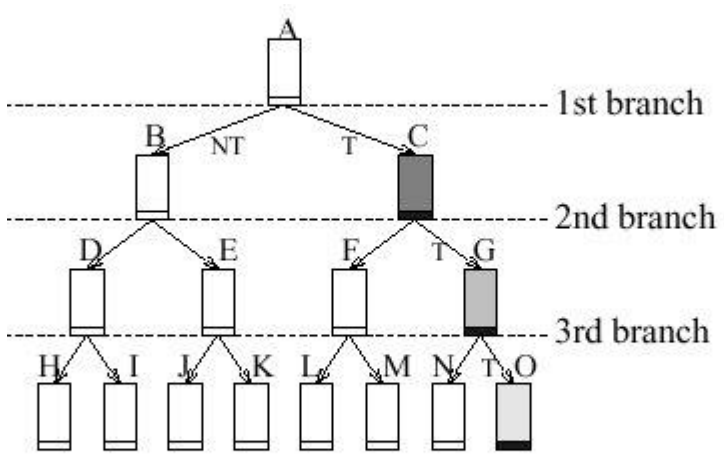
## Basic Blocks/Dynamic Execution Sequence (Trace) Example

Static Program Order



- **A-O = Basic Blocks terminating with conditional branches**
- **The outcomes of branches determine the basic block dynamic execution sequence or <u>trace</u>**

Program Control Flow Graph (CFG)



If all three branches are taken the execution trace will be basic blocks: ACGO

Average Basic Block Size = 5-7 instructions

NT = Branch Not Taken left child

T = Branch Taken right child

# Increasing Instruction-Level Parallelism (ILP)

- **A common way to increase parallelism among instructions is to exploit parallelism among iterations of a loop**
  - **(i.e  Loop Level Parallelism, LLP).**
- **This is accomplished by unrolling the loop either statically by the compiler, or dynamically by hardware, which increases the size of the basic block present.  This resulting larger basic block provides more instructions that can be scheduled or re-ordered by the compiler to eliminate more stall cycles.**
- **In this loop every iteration can overlap with any other iteration. Overlap within each iteration is minimal.**

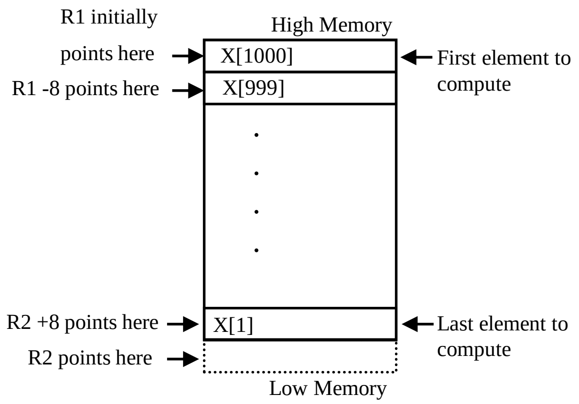
each iteration is independent each other

```
for (i=1; i<=1000; i=i+1;)
              x[i] = x[i] + y[i];
```

# MIPS Loop Unrolling Example

- **For the loop:**

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```



**The straightforward MIPS assembly code is given by:**

```
Loop:  L.D      F0, 0 (R1)      ;F0=array element
       ADD.D    F4, F0, F2      ;add scalar in F2
       S.D      F4, 0(R1)       ;store result
       DADDI    R1, R1, # -8    ;decrement pointer by 8 bytes
       BNE      R1, R2,Loop     ;branch R1!=R2
```

$\boxed{\text{Basic block size } = 5 \text{ instructions}}$

## MIPS FP Latency Assumptions

- **All FP units assumed to be fully pipelined.**
- **The following FP operations latencies are used:**

(or Number of Stall Cycles)

| Instruction Producing Result |  | Instruction Using Result | Latency In Clock Cycles |
|---|---|---|---|
| FP ALU Op | $\longrightarrow$ | Another FP ALU Op | 3 |
| FP ALU Op |  | Store Double | 2 |
| Load Double |  | FP ALU Op | 1 |
| Load Double |  | Store Double | 0 |

# Loop Unrolling Example (continued)

- **This loop code is executed on the MIPS pipeline as follows:**

**(Branch resolved in decode stage)**

No scheduling

```
                                      Clock cycle
Loop:  L.D       F0, 0(R1)                 1
       stall                               2
       ADD.D     F4, F0, F2                3
       stall                               4
       stall                               5
       S.D       F4, 0 (R1)                6
       DADDUI    R1, R1, # -8              7
       stall                               8
       BNE       R1,R2, Loop               9
```

9 cycles per iteration

Scheduled:

```
Loop:  L.D       F0, 0(R1)
       DADDUI    R1, R1, # -8
       ADD.D     F4, F0, F2
       stall
       stall
       S.D       F4,8(R1)
       BNE       R1,R2, Loop
```

9/7 = 1.3 times faster

7 cycles per iteration

# Loop Unrolling Example (continued)

- **The resulting loop code when four copies of the loop body are unrolled without reuse of registers.**
- **<u>The size of the basic block</u> increased from 5 instructions in the original loop to <u>14 instructions</u>.**

Cycle

<u>No scheduling</u>

```
Loop: 1  L.D      F0, 0(R1)
      2  Stall
      3  ADD.D    F4, F0, F2
      4  Stall
      5  Stall
      6  S.D      F4,0 (R1)        ; drop DADDUI & BNE
      7  L.D      F6, -8(R1)
      8  Stall
      9  ADD.D    F8, F6, F2
      10 Stall
      11 Stall
      12 S.D      F8, -8 (R1),     ; drop DADDUI & BNE
      13 L.D      F10, -16(R1)
      14 Stall
      15 ADD.D    F12, F10, F2
      16 Stall
      17 Stall
      18 S.D      F12, -16 (R1)    ; drop DADDUI & BNE
      19 L.D      F14, -24 (R1)
      20 Stall
      21 ADD.D    F16, F14, F2
      22 Stall
      23 Stall
      24 S.D      F16, -24(R1)
      25 DADDUI   R1, R1, # -32
      26 Stall
      27 BNE      R1, R2, Loop
```

Three branches and three decrements of R1 are eliminated.

Load and store addresses are changed to allow 4 DADDUI instructions to be merged into one.

The unrolled loop runs in 27 cycles assuming each L.D has 1 stall cycle, each ADD.D has 2 stall cycles, and the DADDUI 1 stall, or 27/4 = 6.75 cycles to produce each of the four iterations.

# Loop Unrolling Example (continued)

**When scheduled for pipeline**

```
Loop:  L.D      F0, 0(R1)
       L.D      F6,-8 (R1)
       L.D      F10, -16(R1)
       L.D      F14, -24(R1)
       ADD.D    F4, F0, F2
       ADD.D    F8, F6, F2
       ADD.D    F12, F10, F2
       ADD.D    F16, F14, F2
       S.D      F4, 0(R1)
       S.D      F8, -8(R1)
       DADDUI   R1, R1,# -32
       S.D      F12, 16(R1)
       S.D      F16, 8(R1);8-32 = -24
       BNE      R1,R2, Loop
```

The execution time of the loop has dropped to 14 cycles, or 14/4 = 3.5 clock cycles per iteration

Unrolling the loop exposed more computations that can be scheduled to minimize stalls by increasing the size of the basic block from 5 instructions in the original loop to 14 instructions in the unrolled loop.

# Loop Unrolling Benefits & Requirements

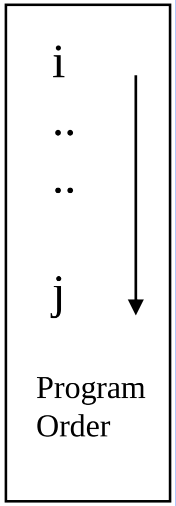
- **Loop unrolling improves performance in two ways:**
  - Larger basic block size: More instructions to schedule and thus possibly more stall cycles are eliminated.
  - Fewer instructions executed: Fewer branches and loop maintenance instructions executed
- **From the loop unrolling example, the following guidelines were followed:**
  - Determine that unrolling the loop would be useful by finding that the loop iterations were independent.
  - Determine that it was legal to move S.D after DADDUI; find the correct S.D offset.
  - Use different registers (rename registers) to avoid constraints of using the same registers (WAR, WAW).
  - Eliminate extra tests and branches and adjust loop maintenance code.
  - Determine that loads and stores can be interchanged by observing that they are independent in different loops.
  - Schedule the code, preserving any dependencies needed to give the same result as the original code.

# Instruction Dependencies

- **Determining instruction dependencies (dependence analysis) is important for pipeline scheduling and to determine the amount of instruction level parallelism (ILP) in the program to be exploited.**
- **Instruction Dependence Graph: A directed graph where nodes represent instructions and edges represent instruction dependencies.**
- **If two instructions are independent or parallel (no dependencies between them exist), they can be executed simultaneously in the pipeline without causing stalls (no pipeline hazards); assuming the pipeline has sufficient resources (no structural hazards).**
- **Instructions that are dependent are not parallel and cannot be reordered by the compiler or hardware.**
- **Instruction dependencies are classified as:**
  - **Data dependencies**
  - **Name dependencies** ← Name: Register or Memory Location
  - **Control dependencies**

# Name Dependencies

- A name dependence occurs when two instructions use (share) the same register or memory location, called ***a name***.
- No flow of data exist between the instructions involved in the name dependence (i.e. no producer/consumer relationship)
- If instruction *i* precedes instruction *j* in program order then two types of name dependencies can exist:



- An anti-dependence exists when *j* writes to the same register or memory location that instruction *i* reads
  - Anti-dependence violation: Relative read/write order is changed
    - This results in a WAR hazard and thus the relative instruction read/write and execution order must be preserved.
- An output or (write) dependence exists when instruction *i* and *j* write to the same register or memory location
  - Output-dependence violation: Relative write order is changed
    - This results in a WAW hazard and thus instruction write and execution order must be preserved

# Instruction Dependence Example

- **For the following code identify all data and name dependencies between instructions and give the dependence 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):

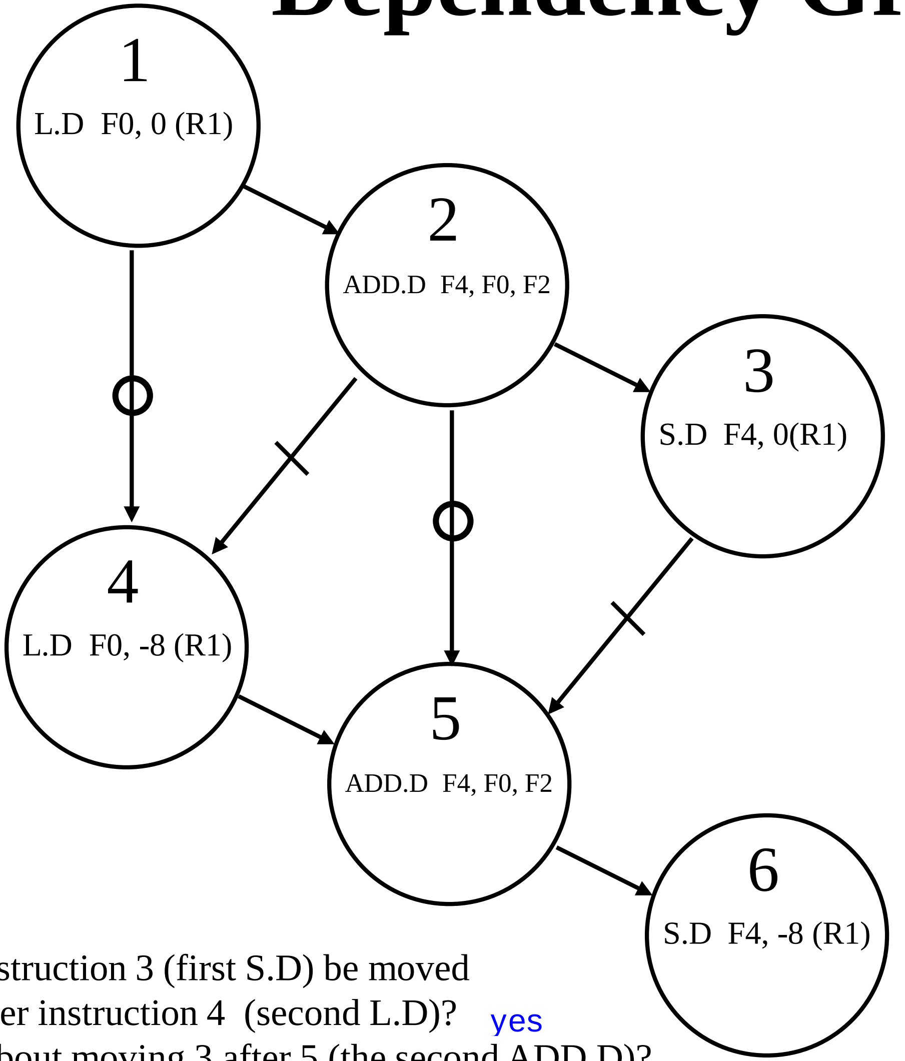
Instruction 1 has an output name dependence (WAW) over result register (name) F0 with instructions 4
Instruction 2 has an output name dependence (WAW) over result register (name) F4 with instructions 5

### 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)

Can instruction 4 (second L.D) be moved just after instruction 1 (first L.D)?
If not what dependencies are violated?

Can instruction 3 (first S.D) be moved just after instruction 4 (second L.D)? yes
How about moving 3 after 5 (the second ADD.D)?
If not what dependencies are violated? no