CMSC 132: Computer Architecture

Asst. Prof. Reginald Neil C. Recario
rncrecario@gmail.com
Institute of Computer Science
University of the Philippines Los Baños

Instruction-Level Parallelism and Its Exploitation

- What is Instruction-Level Parallelism?

Instruction-Level Parallelism and Its Exploitation

- **Instruction-Level Parallelism (ILP)**
  - The potential overlap among instructions

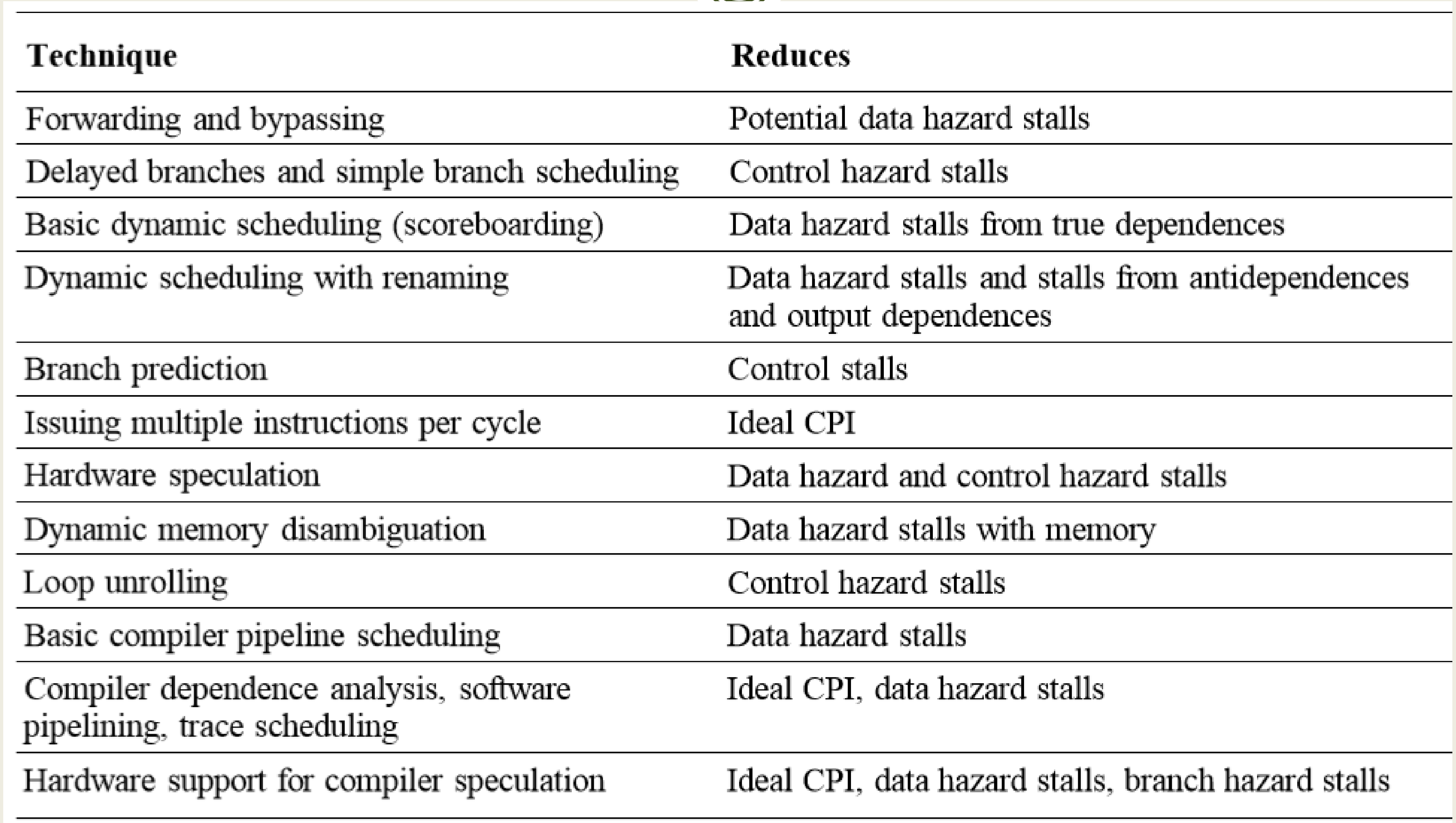
- The value of the CPI for a pipelined processor is the sum of the base CPI and all contributions from stalls.

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

Instruction-Level Parallelism and Its Exploitation

- Ideal pipeline CPI is the measure of the maximum performance attainable by the implementation.

Instruction-Level Parallelism and Its Exploitation



Technique	Reduces
Forwarding and bypassing	Potential data hazard stalls
Delayed branches and simple branch scheduling	Control hazard stalls
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences
Dynamic scheduling with renaming	Data hazard stalls and stalls from antidependences and output dependences
Branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI
Hardware speculation	Data hazard and control hazard stalls
Dynamic memory disambiguation	Data hazard stalls with memory
Loop unrolling	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls

Figure 2.1 The major techniques examined in Appendix A, Chapter 2, or Appendix G are shown together with the component of the CPI equation that the technique affects.

Instruction-Level Parallelism and Its Exploitation

- **Basic block** – a straight line of code sequence with no branches in except to the entry and no branches out except at the exit

- Amount of parallelism available within a basic block is quite small.

Instruction-Level Parallelism and Its Exploitation

- The simplest and most common way to increase the ILP is to exploit parallelism among iterations of a loop. **This is called loop-level parallelism**.

```
for(i=1; i<=100; i++)
    x[i] = x[i] + y[i];
```

Data Dependencies and Hazards

- To exploit instruction-level parallelism we must determine which instructions can be executed in parallel.
- If two instructions are *parallel*, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls, assuming the pipeline has sufficient resources (and hence no structural hazards exist).

Data Dependencies and Hazards

- If two instructions are *dependent*, they are not parallel and must be executed in order, although they may often be partially overlapped. The key in both cases is to determine whether an instruction is dependent on another instruction.

Data Dependencies and Hazards

- There are three different types of dependencies: **data dependencies** (also called **true data dependencies**), **name dependencies**, and **control dependencies**.

Data Dependencies

- **Data Dependencies**
- An instruction *j* is ***data dependent*** on instruction *i* if either of the following holds:
  - Instruction *i* produces a result that may be used by instruction *j*, or
  - Instruction *j* is data dependent on instruction *k*, and instruction *k* is data dependent on instruction *i*.

Data Dependencies

- Example:

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

Data Dependencies

- The data dependencies in the code sequence involve both floating-point data:

```
Loop:   L.D F0, 0(R1)
        ADD.D F4, F0, F2
        S.D F4, 0(R1)
        ...
```

Data Dependencies

- Data dependency in integer data:

```
DADDUI R1, R1,-8
BNE R1, R2, Loop
...
```

Data Dependencies

- If two instructions are **data dependent**, they cannot execute simultaneously or be completely overlapped. The dependence implies that there would be a chain of one or more data hazards between the two instructions.

Data Dependencies

- Executing the instructions simultaneously will cause a processor with pipeline to detect a hazard and stall, thereby reducing or eliminating the overlap. In a processor without interlocks that relies on compiler scheduling, the compiler cannot schedule dependent instructions in such a way that they completely overlap, since the program will not execute correctly.

Data Dependencies

- The presence of a data dependence in an instruction sequence reflects a data dependence in the source code from which the instruction sequence was generated.
- The effect of the original data dependence must be preserved.
- Dependencies are a property of programs.

Data Dependencies

- A data dependency conveys three things:
  - (1) the possibility of a hazard
  - (2) the order in which results must be calculated, and
  - (3) an upper bound on how much parallelism can possibly be exploited.

Data Dependencies

- A dependence can be overcome in two different ways:
  - maintaining the dependence but avoiding a hazard, and
  - eliminating a dependence by transforming the code.

Data Dependencies

- Scheduling the code is the primary method used to avoid a hazard without altering a dependence, and such scheduling can be done both by the compiler and by the hardware.

Data Dependencies

- A data value may flow between instructions either through registers or through memory locations
  - Easier to detect if through registers

Name Dependencies

- **Name Dependencies**
- A ***name dependence*** occurs when two instructions use the same register or memory location, called a *name*, but there is no flow of data between the instructions associated with that name.

Name Dependencies

- There are two types of name dependencies between an instruction *i* that precedes instruction *j* in program order:

Name Dependencies

- An **antidependence** between instruction *i* and instruction *j* occurs when instruction *j* **writes** a register or memory location that instruction *i* reads. The original ordering must be preserved to ensure that *i* reads the correct value.

  Example:

```
S.D F4, 0(R1)
DADDUI R1, R1,-8
```

Name Dependencies

- **An output dependence** occurs when instruction *i* and instruction *j* write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction *j*.

Name Dependencies

- Since a name dependence is not a true (data) dependence, instructions involved in a name dependence can execute simultaneously or be reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict.

Name Dependencies

- This renaming can be more easily done for register operands, where it is called **register renaming**.
- **Register renaming** can be done either statically by a compiler or dynamically by the hardware.

Data Hazards

- A **hazard** is created whenever there is a dependence between instructions, and they are close enough that the overlap during execution would change the order of access to the operand involved in the dependence.

Data Hazards

- Because of the dependence, we must preserve what is called *program order*, that is, the order that the instructions would execute in if executed sequentially one at a time as determined by the original source program.

Data Hazards

- The goal of both our software and hardware techniques is to exploit parallelism by preserving program order only where it affects the outcome of the program.
- Detecting and avoiding hazards ensures that necessary program order is preserved.

Data Hazards

- Data hazards may be classified as one of three types, depending on the order of read and write accesses in the instructions. By convention, the hazards are named by the ordering in the program that must be preserved by the pipeline.

Data Hazards

- The possible data hazards are
  - **RAW** (Read After Write)
  - **WAW** (Write After Write)
  - **WAR** (Write After Read)

Data Hazards

- **RAW (read after write)** – *j* tries to read a source before *i* writes it, so *j* incorrectly gets the old value. This hazard is the most common type and corresponds to a true data dependence. Program order must be preserved to ensure that *j* receives the value from *i*.

Data Hazards

- **WAW (write after write)** — *j* tries to write an operand before it is written by *i*. The writes end up being performed in the wrong order, leaving the value written by *i* rather than the value written by *j* in the destination.

Data Hazards

- WAW hazard corresponds to an output dependence. WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.

Data Hazards

- **WAR (write after read)** — *j* tries to write a destination before it is read by *i*, so *i* incorrectly gets the new value. This hazard arises from an antidependence.

Data Hazards

- WAR hazards cannot occur in most static issue pipelines—even deeper pipelines or floating-point pipelines—because all reads are early (in ID) and all writes are late (in WB).
- A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline and other instructions that read a source late in the pipeline, or when instructions are reordered.

Data Hazards

- High-level PL example*:

RAW

```
a = b + c;
f = a + r;
```

WAW

```
a = b + c;
a = f + r;
```

WAR

```
f = a + r;
a = b + c;
```

**Note: These instructions however is composed of more than one microinstruction.*

Control Dependencies

- **Control Dependencies**
- The last type of dependence is a control dependence. A **control dependence** determines the ordering of an instruction, *i*, with respect to a branch instruction so that the instruction *i* is executed in correct program order and only when it should be.

Control Dependencies

- Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order.
- One of the simplest examples of a control dependence is the dependence of the statements in the "*then*" part of an *if statement* on the branch.

Control Dependencies

- Example:

```
if P1 {
     S1;
  };
if P2 {
       S2;
} ;
```

- S1 is control dependent on P1, and S2 is control dependent on P2 but not on P1.

Control Dependencies

- In general, there are two constraints imposed by control dependencies:
  - An instruction that is control dependent on a branch cannot be moved *before* the branch so that its execution *is no longer controlled* by the branch. For example, we cannot take an instruction from the *then* portion of an if statement and move it before the *if statement*.

Control Dependencies

- An instruction that is not control dependent on a branch cannot be moved *after* the branch so that its execution *is controlled* by the branch. For example, we cannot take a statement before the *if statement* and move it into the *then* portion.

Control Dependencies

- When processors preserve strict program order, they ensure that control dependencies are also preserved.
- Control dependence is not the critical property that must be preserved.
- Two properties critical to program correctness are the **exception behavior** and the **data flow**.

Control Dependencies

- **Preserving the exception behavior**
  - Any changes in the ordering of instruction execution must not change how exceptions are raised in the program.
  - Often this is relaxed to mean that the reordering of instruction execution must not cause any new exceptions in the program.

Control Dependencies

- A simple example shows how maintaining the control and data dependencies can prevent such situations.

```
   DADDU R2,R3,R4
   BEQZ R2.L1
   LW R1,0(R2)
LI:
```

Control Dependencies

- **Dataflow**
  - It is the actual flow of data values among instructions that produce results and those that consume them.
  - Branches make the data flow dynamic, since they allow the source of data for a given instruction to come from many points.
  - Program order is ensured by maintaining the control dependencies.

Control Dependencies

```
  DADDU R1.R2.R3
  BEQZ R4,L
  DSUBU R1.R5.R6
L: ...
  OR R7,R1,R8
```

Control Dependencies

- Sometimes we can determine that violating the control dependence cannot affect either the exception behavior or the data flow.

Control Dependencies

- Example:

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

Control Dependencies

- Assume DSUBU R4 is unused (register).
  - If it is *dead*, then we can move it out of the code region *before* the branch.
- The property of whether a value will be used by an upcoming instruction is called **liveness**.

Control Dependencies

- Control dependence is preserved by implementing control hazard detection that causes control stalls.
- Control stalls can be eliminated or reduced by a variety of hardware and software techniques.
- [Other solution would involve speculation (**hardware** and **software**).]

Basic Compiler Techniques for Exposing ILP

- How does the compiler expose ILP?
  - Basic Pipeline Scheduling
  - Loop Unrolling

Basic Pipeline Scheduling and Loop Unrolling

- To keep a pipeline full, parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline.
- To avoid a pipeline stall, a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction.

Basic Pipeline Scheduling and Loop Unrolling

- A compiler's ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the functional units in the pipeline.

Basic Pipeline Scheduling and Loop Unrolling

- We assume the standard five-stage integer pipeline, so that branches have a delay of 1 clock cycle. We assume that the functional units are fully pipelined or replicated (as many times as the pipeline depth), so that an operation of any type can be issued on every clock cycle and there are no structural hazards.

Basic Pipeline Scheduling and Loop Unrolling

- We will rely on the following code segment, which adds a scalar to a vector:

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

- We can see that this loop is parallel by noticing that the body of each iteration is independent.

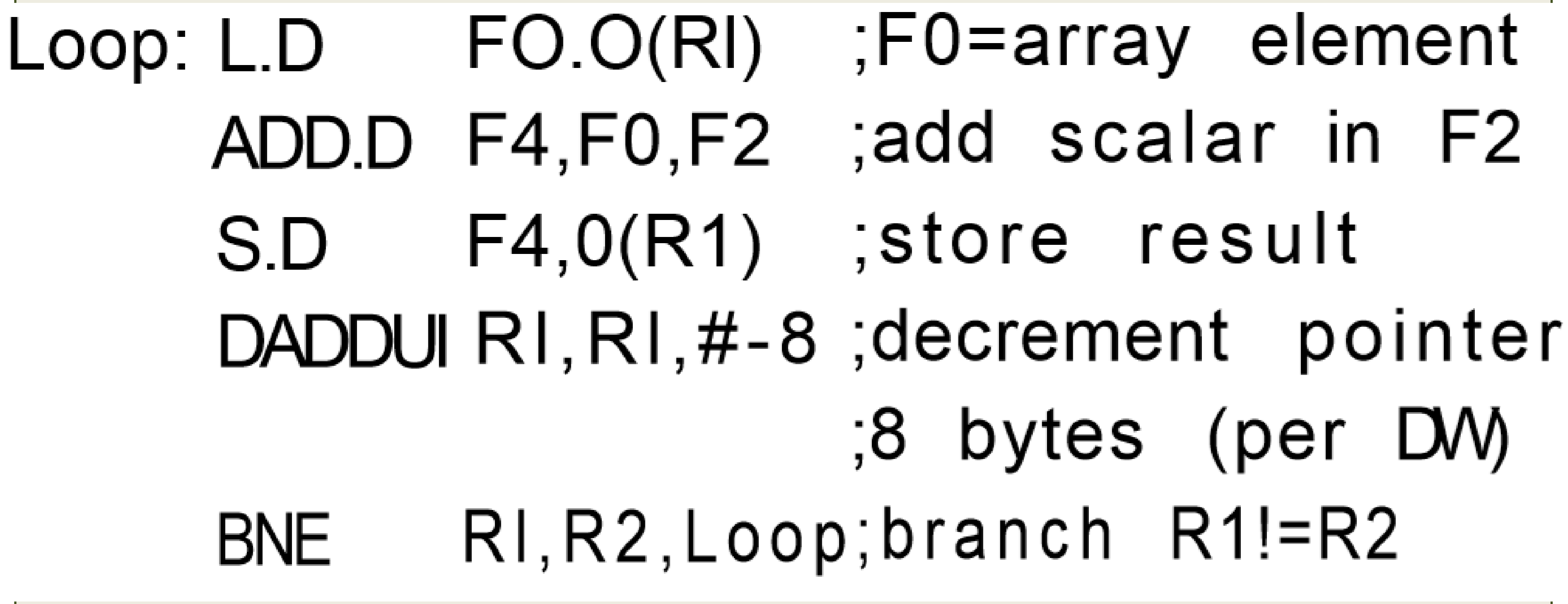
Basic Pipeline Scheduling and Loop Unrolling

- The first step is to translate the above segment to MIPS assembly language.
- In the following code segment, R1 is initially the address of the element in the array with the highest address, and F2 contains the scalar values.

Basic Pipeline Scheduling and Loop Unrolling

- Register R2 is precompiled, so that 8(R2) is the address of the last element to operate on. The straightforward MIPS code, not scheduled for the pipeline, looks like this:

Basic Pipeline Scheduling and Loop Unrolling



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

Basic Pipeline Scheduling and Loop Unrolling

- **Example:**

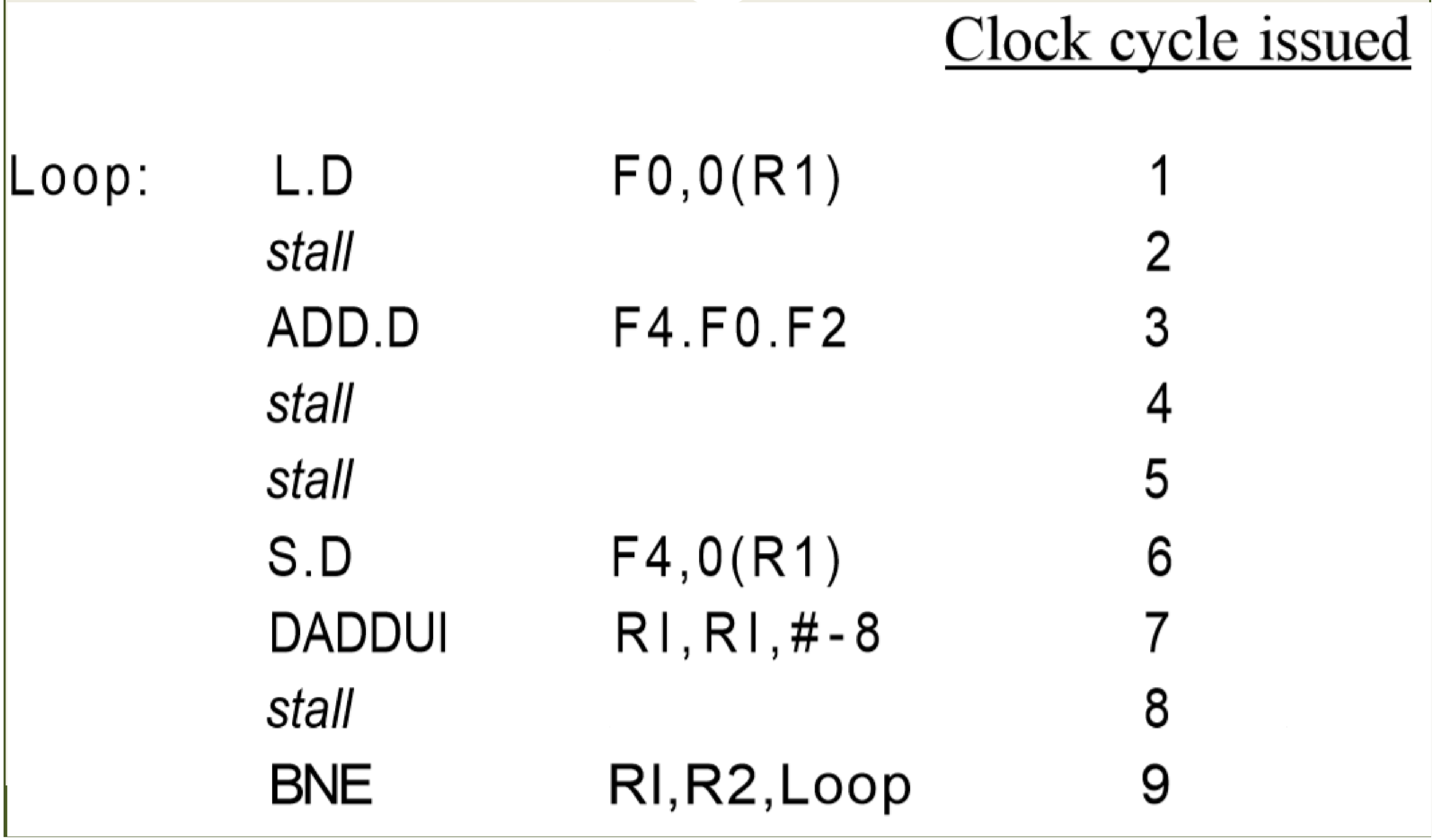
  Show how the loop would look on MIPS, both scheduled and unscheduled, including any stalls or idle clock cycles. Schedule for delays from floating-point operations, but remember that we are ignoring delayed branches.

Basic Pipeline Scheduling and Loop Unrolling

- **Answer:**

  Without any scheduling, the loop will execute as follows, taking 9 cycles:

Basic Pipeline Scheduling and Loop Unrolling

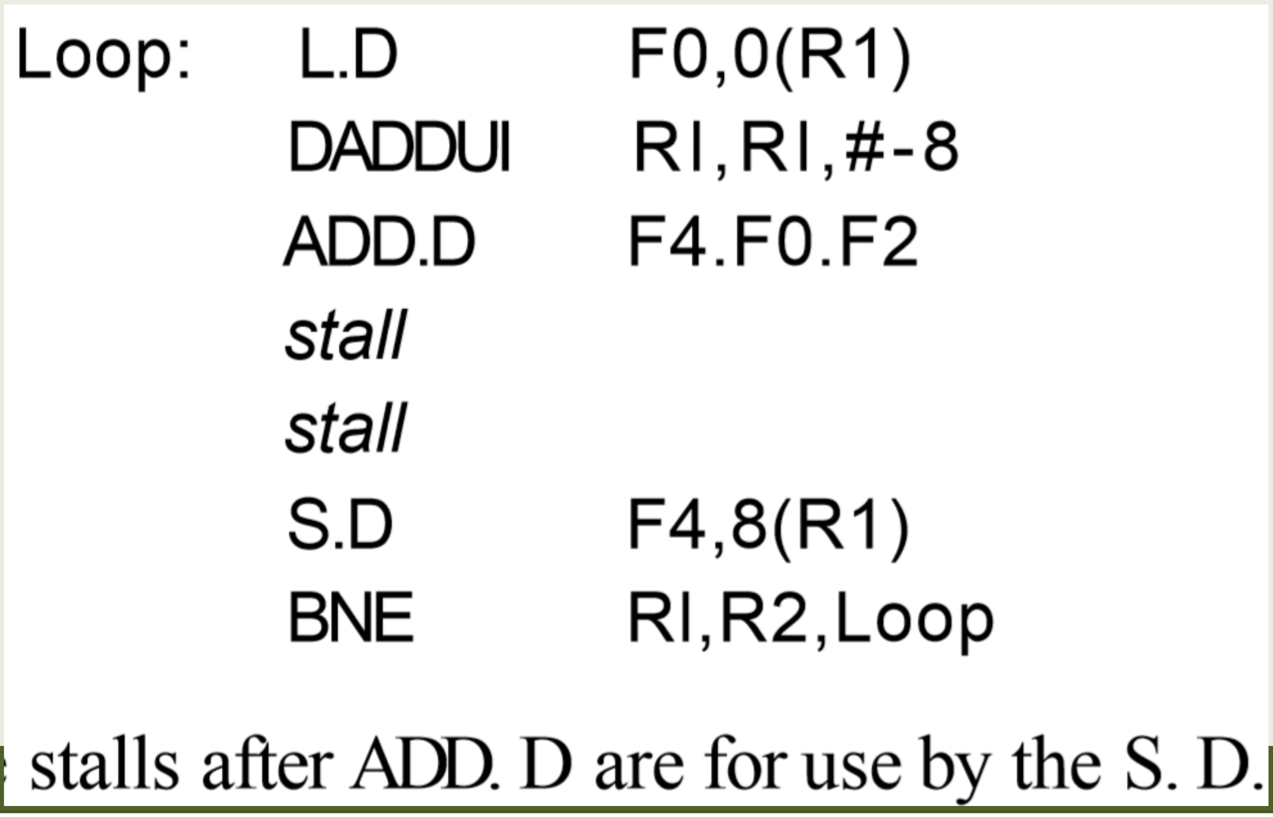


```
                                                  Clock cycle issued

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      RI,RI,#-8                            7
        stall                                            8
        BNE         RI,R2,Loop                           9
```

Basic Pipeline Scheduling and Loop Unrolling

- We can schedule the loop to obtain only two stalls and reduce the time to 7 cycles:



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

stalls after ADD. D are for use by the S. D.

Basic Pipeline Scheduling and Loop Unrolling

- A simple scheme for increasing the number of instructions relative to the branch and overhead instructions is **loop unrolling**.
- Unrolling simply replicates the loop body multiple times, adjusting the loop termination code.

Basic Pipeline Scheduling and Loop Unrolling

- Loop unrolling can also be used to improve scheduling. Because it eliminates the branch, it allows instructions from different iterations to be scheduled together..

Basic Pipeline Scheduling and Loop Unrolling

- In this case, we can eliminate the data use stalls by creating additional independent instructions within the loop body.

Basic Pipeline Scheduling and Loop Unrolling

- If we *simply* replicated the instructions when we unrolled the loop, the resulting use of the same registers could prevent us from effectively scheduling the loop.
- Thus, we will want to use different registers for each iteration, increasing the required number of registers.

Basic Pipeline Scheduling and Loop Unrolling

- **Example:**

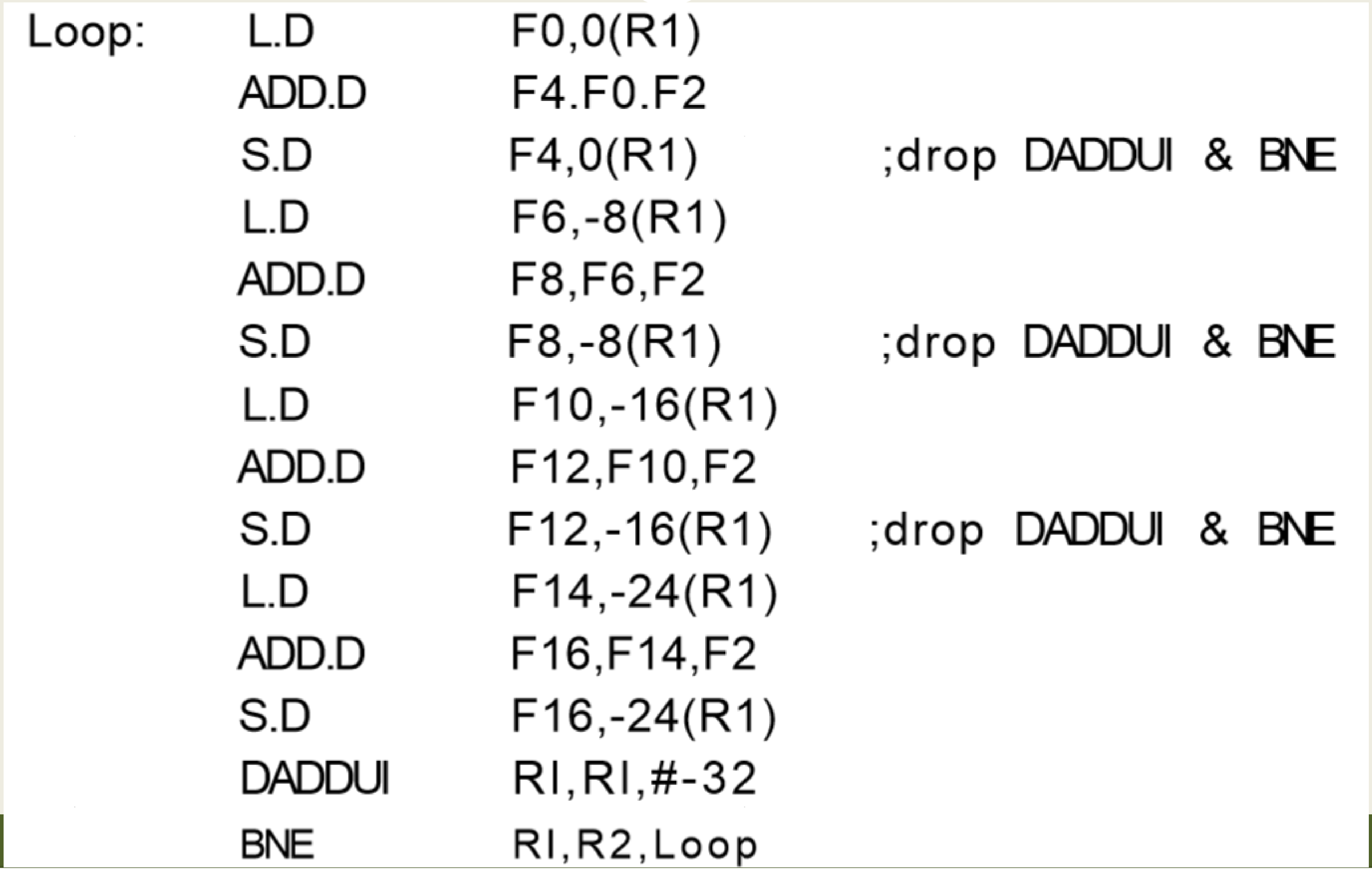
  Show our loop unrolled so that there are four copies of the loop body, assuming R1 - R2 (that is, the size of the array) is initially a multiple of 32, which means that the number of loop iterations is a multiple of 4. Eliminate any obviously redundant computations and do not reuse any of the registers.

Basic Pipeline Scheduling and Loop Unrolling

- **Answer:**

  Here is the result after merging the DADDUI instructions and dropping the unnecessary BNE operations that are duplicated during unrolling. Note that R2 must now be set so that 32 (R2) is the starting address of the last four elements.

Basic Pipeline Scheduling and Loop Unrolling



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

Basic Pipeline Scheduling and Loop Unrolling

- In real programs we do not usually know the upper bound on the loop. Suppose it is n, and we would like to unroll the loop to make k copies of the body.
- How will the unrolled loop perform when it is scheduled for the pipeline described earlier?

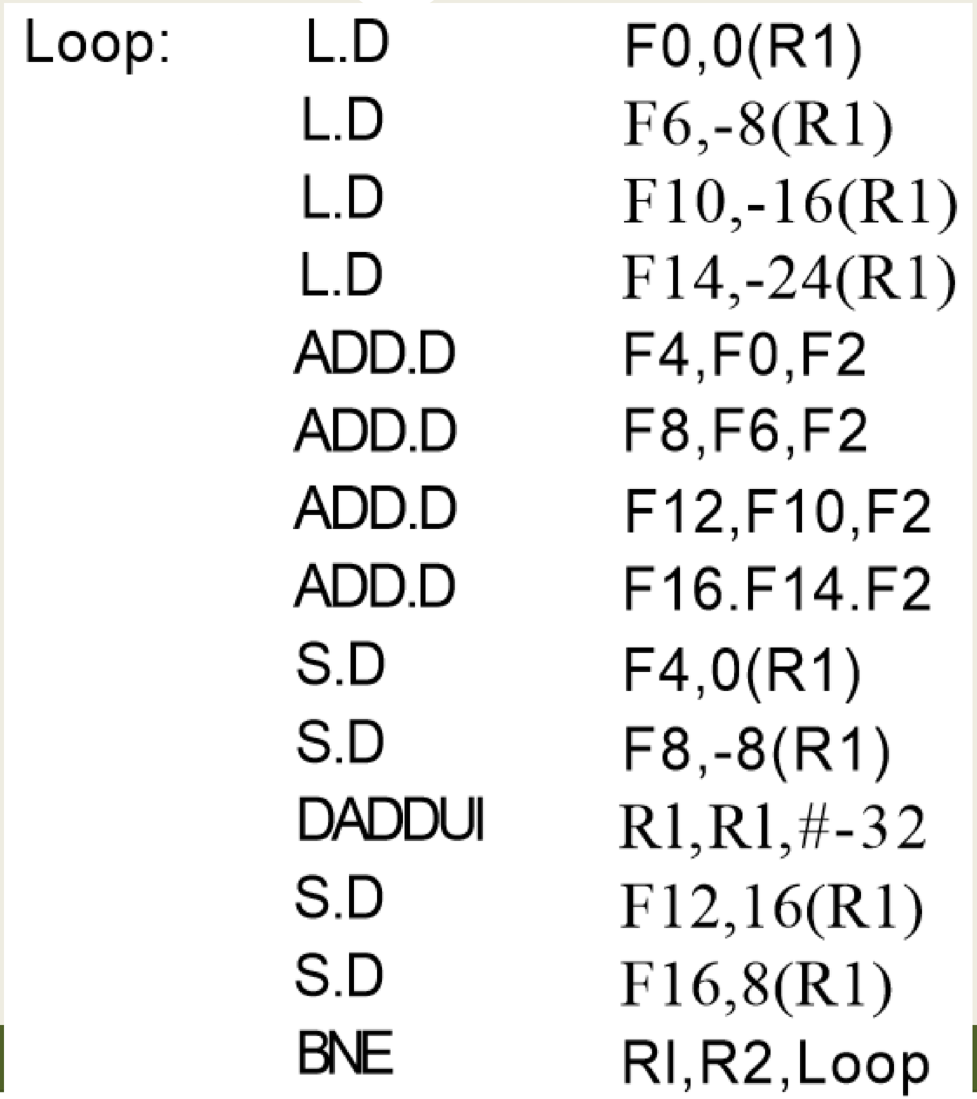
Basic Pipeline Scheduling and Loop Unrolling

- **Example**:

  Show the unrolled loop in the previous example after it has been scheduled for the pipeline with the latencies.

Basic Pipeline Scheduling and Loop Unrolling

- **Answer:**



```
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)
        BNE       RI,R2,Loop
```

Basic Pipeline Scheduling and Loop Unrolling

- **Answer:**

  The execution time of the unrolled loop has dropped to a total of 14 clock cycles, or 3.5 clock cycles per element, compared with 9 cycles per element before any unrolling or scheduling and 7 cycles when scheduled but not unrolled.

Basic Pipeline Scheduling and Loop Unrolling

- Summary:
- To obtain the final unrolled code we had to make the following decisions and transformations:
  - Determine that unrolling the loop would be useful by finding that the loop iterations were independent, except for the loop maintenance code.
  - Use different registers to avoid unnecessary constraints that would be forced by using the same registers for different computations.

.

Basic Pipeline Scheduling and Loop Unrolling

- Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.

Basic Pipeline Scheduling and Loop Unrolling

- Determine that the loads and stores in the unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent. This transformation requires analyzing the memory addresses and finding that they do not refer to the same address.
- Schedule the code, preserving any dependences needed to yield the same result as the original code

Reference(s):

- **Hennessy, J.L., Patterson, D.A**. Computer Architecture: A Quantitative Approach (4th Ed)