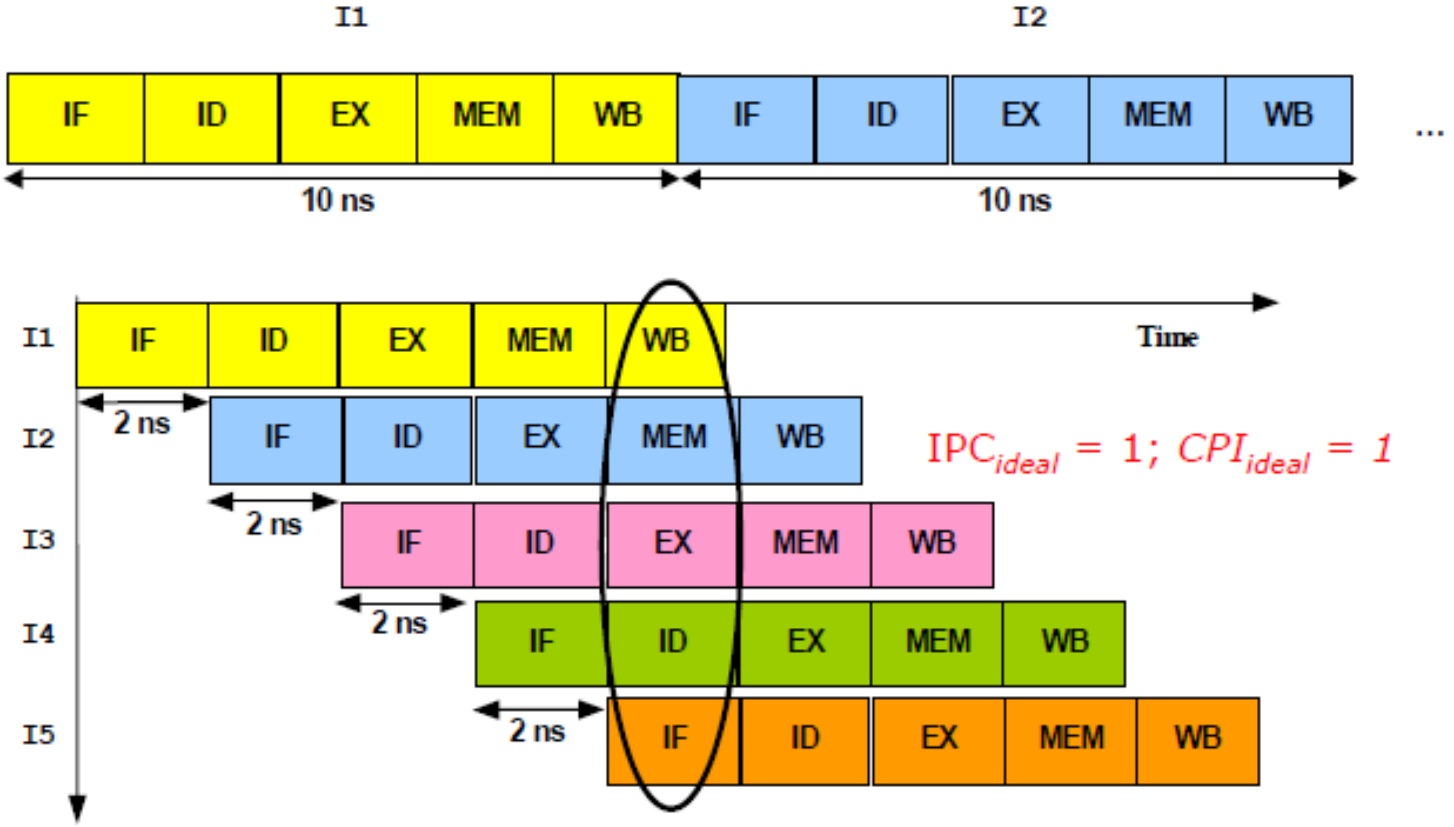
Instruction Level Parallelism

Instruction Level Parallelism

- Instructions evaluated in parallel
- Sequential vs. Pipelining Execution



Instruction Level Parallelism

- Two approaches to exploit ILP
  - Hardware based - dynamic
  - Software (compiler) based - static
- In a pipelined machine, actual CPI is derived as:

$$CPI_{Pipeline} = CPI_{Ideal} + \text{Structural stalls} + \text{Data hazard stalls} + \text{Control stalls}$$

- Reduction of any right-hand term reduces $CPI_{pipeline}$ *to* CPI_{ideal} or alternatively increase the Instructions Per Clock

$$IPC = 1 / CPI$$

- Best case: the max throughput would be to complete 1 Instruction Per Clock:

$$IPC_{ideal} = 1;\ CPI_{ideal} = 1$$

Instruction Level Parallelism

- For typical MIPS programs, the average dynamic branch frequency is often between 15% and 25%.
- Since these instructions are likely to depend upon one another, the amount of overlap that can be exploited within a basic block is likely to be less than the average basic block size.
- To obtain substantial performance enhancements, ILP must be exploited across multiple basic blocks
- Best way to increase ILP is to exploit parallelism among iterations of a loop - loop-level parallelism.

Instruction Level Parallelism

- Example – Loop level parallelism

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

- Computation in each iteration is independent of the previous iterations and the loop is thus parallel

- Different techniques to convert loop-level parallelism into Instruction level parallelism.

Dependences

- Determining **dependences** among instructions is critical to defining the amount of parallelism existing in a program
- To exploit instruction-level parallelism, it is critical to 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).
- If two instructions are dependent, they are not parallel and must be executed in order, although they may often be partially overlapped

Dependences

- Three different types of dependences:
  - *data dependences*
  - *name dependences*
  - *control dependences*

- An instruction j is data dependent on instruction i if:

  - *Instruction i* produces a result that may be used by *instruction j*.

  - *Instruction j* is data dependent on instruction k, and *instruction k* is data dependent on instruction i.

Data Dependences

- Example

```
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

      DADDUI R1,R1,#-8 ;decrement pointer 8 bytes

      BNE R1,R2,LOOP    ;branch R1!=R2
```

Data Dependences

- Example

```
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
      DADDUI R1,R1,#-8   ;decrement pointer 8 bytes
      BNE R1,R2,LOOP     ;branch R1!=R2
```

Data Dependences

- A data dependence conveys three things:
  - possibility of a hazard
  - order in which results must be calculated
  - an upper bound on parallelism possibly be exploited

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

- 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.

- A data value may flow between instructions either through registers or through memory locations - detecting dependence is straightforward since the register names are fixed in the instructions.

Name Dependences

- 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.

- Two types of name dependences between an instruction *i* that *precedes* instruction *j* in program order:

  - *Antidependence* - 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.

```
i: sub r4,r1,r3
j: add r1,r2,r3
k: mul r6,r1,r7
```

Name Dependences

- *Output dependence* - 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.

```
i: sub r1,r4,r3
j: add r1,r2,r3
k: mul r6,r1,r7
```

Name Dependences

- 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.

- 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 exists whenever there is a name or data 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.

- *program order*— the order that the instructions would execute in if executed sequentially one at a time as determined by the original source program.

- The goal of 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

- Consider two instructions *i* and *j,* with *i* preceding *j* in program order. The possible data hazards are:
  - RAW *(read after write)*—*j* tries to read a source before *i* writes it, so *j* incorrectly gets the *old* value.
  - WAW *(write after write)* — *j* tries to write an operand before it is written by *i*.
  - WAR (write after read)—*j* tries to write a destination before it is read by *i*, so *i* incorrectly gets the new value.

RAW

- RAW *(read after write)*—*j* tries to read a source before *i* writes it, so *j* incorrectly gets the *old* value.

  corresponds to a true data dependence. Program order must be preserved to ensure that *j* receives the value from *i.*

Example

```
DADD R1, R2, R3
DSUB R4, R5, R1
```

- Subtract reads output of the addition, creating a RAW hazard

WAW

- WAW *(write after write)* — *j* tries to write an operand before it is written by *i*.

  The writes end up in the wrong order, leaving the value written by *i* rather than the value written by *j* in the destination. Corresponds to an output dependence. Occur only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.

Example

```
DADD R1, R2, R3
DSUB R1, R5, R6
```

- Subtract writes the same register as the addition, creating a WAW hazard

WAR

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

  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

Example

```
DADD R1, R2, R3
DSUB R2, R5, R6
```

- Subtract writes R2, which is read by the addition, creating a WAR hazard

Data Hazards

- A data/name dependence can potentially generate a data hazard (RAW, WAW, or WAR), but the actual hazard and the number of stalls to eliminate the hazards are a property of the pipeline.

- Dependences are a property of the program, while hazards are a property of the pipeline.

Control Dependences

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

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

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

Control Dependences

- Two constraints are imposed by control dependences
  - 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.
  - 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.
- When processors preserve strict program order, they ensure that control dependences are also preserved.
- Two properties critical to program correctness—and normally preserved by maintaining both data and control dependences—are the exception behavior and the data flow.

Control Dependences

- Example

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

  - if control dependence is ignored and load instruction is moved before the branch, the load instruction may cause a memory protection exception.
  - no data dependence prevents from interchanging BEQZ and LW; it is only the control dependence
  - data flow 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.

Control Dependences

- Example

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

  - Value of R1 used by the OR instruction depends on whether the branch is taken or not.
  - Data dependence alone is not sufficient to preserve correctness.
  - The OR instruction is data dependent on both the DADDU and DSUBU instructions, but preserving that order alone is insufficient for correct execution.
  - the DSUBU instruction cannot be moved above the branch

Basic Compiler Techniques for Exposing ILP

Basic Pipeline Scheduling

- 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, the execution of 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.

- 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

- Assumption
  - standard five-stage integer pipeline, so that branches have a delay of one clock cycle.
  - 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.
- **Latencies of FP operations**

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Basic Pipeline Scheduling

- Consider the following code segment, which adds a scalar to a vector:

```
for (i=999; i>=0; i=i-1)
x[i] = x[i] + s;
```

- Loop is parallel - the body of each iteration is independent

```
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
      DADDUI R1,R1,#-8    ;decrement pointer
                          ;8 bytes (per DW)
      BNE R1,R2,Loop      ;branch R1!=R2
```

Basic Pipeline Scheduling

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, ignoring delayed branches.

Without Scheduling

			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	R1,R1,#-8	7
	stall		8
	BNE	R1,R2,Loop	9

With Scheduling

```
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
```

Loop Unrolling

- simple scheme for increasing the number of instructions relative to the branch and overhead instructions
- Unrolling replicates the loop body multiple times, adjusting the loop termination code.
- used to improve scheduling – eliminates the branch, it allows instructions from different iterations to be scheduled together
- Eliminate the data use stalls by creating additional independent instructions within the loop body.

Loop Unrolling

- Example: Unroll the loop 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.

```
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  R1,R1,#-32
        BNE     R1,R2,Loop
```

Loop Unrolling

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

```
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      R1,R2,Loop
```

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.

Loop Unrolling

- Limitations
  - code size limitations
  - compiler limitations

Branch Prediction