Advanced Pipelining and ILP

Review: Summary of pipelining Basics

- Hazards limit performance
  - Structural: need more hardware
  - Data: need forwarding, compiler rescheduling
  - Control: early evaluation PC, delayed branch, prediction
- Increasing length of pipeline increases impact of hazards
- Pipelining helps instruction bandwidth, not latency
- Interrupts, Floating Point Instructions make pipelining harder
- Compilers reduce cost of data and control hazards (e.g. load delay slots)
- Longer pipelines (R4000): More instruction parallelism, Needs Better branch prediction

Advanced Pipelining and Instruction Level Parallelism (ILP)

- The potential overlap among instructions is called Instruction-Level Parallelism (ILP)
- ILP: Overlap execution of unrelated instructions
- Looking at techniques that reduce the impact of data and control hazards (software approaches)
  - gcc 17% control transfer
    - » 5 instructions + 1 branch
    - » Beyond single block to get more instruction level parallelism
- Increasing the ability of the processor to exploit parallelism (hardware approaches)

CPI of a Pipelined machine

- Pipeline CPI

  = Ideal pipeline CPI + Structural Stalls + Data stalls + Control stalls

  = Ideal pipeline CPI + Structural Stalls + (RAW stalls + WAR stalls + WAW stalls) + Control stalls

Technique	Reduces
Loop Unrolling	Control stalls
Basic Pipeline Scheduling	RAW stalls
Dynamic scheduling with scoreboarding	RAW stalls
Dynamic scheduling with register renaming	WAR and WAW stalls
Dynamic branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI

Loop Unrolling

- The simplest and most common way to increase the parallelism among instruction is to exploit parallelism among iterations of a loop
- E.g.

```
For (i=1;i<1000;i=i+1)
    x[i]=x[i]+s;
end
```

- Increase the amount of available instruction-level parallelism: Unrolling Loops

Unrolling Loops

```
For (i=1;i<1000;i=i+1)
   x[i]=x[i]+s;
end
```

- Translate the above code to DLX:
  - Assume R1 is the address of the element in the array with the highest address.
  - F2 contains the scalar value s.
  - Each element in the array contains double-word
  - For simplicity, we assume that the element with the lowest address is at zero

Unrolling Loops: Where are the Hazards?

```
Loop:  LD    F0,0(R1)   ;F0=vector element
       ADDD  F4,F0,F2   ;add scalar from F2
       SD    0(R1),F4   ;store result
       SUBI  R1,R1,8    ;decrement pointer by 8 bytes (DW)
       BNEZ  R1,Loop    ;branch R1!=zero
       NOP              ;delayed branch slot
```

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
Integer op	Integer op	0

- **Where are the stalls?**

FP Loop Showing Stalls

```
1  Loop: LD    F0,0(R1)    ;F0=vector element
2        stall
3        ADDD  F4,F0,F2    ;add scalar in F2
4        stall
5        stall
6        SD    0(R1),F4    ;store result
7        SUBI  R1,R1,8     ;decrement pointer 8B (DW)
8        stall             ;BNEZ reads the operand in ID
9        BNEZ  R1,Loop     ;branch R1!=zero
10       stall             ;delayed branch slot
```

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
Integer op	Integer op	0

- **Rewrite code to minimize stalls?**

Revised FP Loop Minimizing Stalls

```
1 Loop: LD    F0,0(R1)
2       SUBI  R1,R1,8
3       ADDD  F4,F0,F2
4       stall
5       BNEZ  R1,Loop
6       SD    8(R1),F4
```

Notice: changing address of SD

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
Integer op	Integer op	0

- **6 clock cycles per iteration**
- **Unroll loop 4 times code to make it faster?**

Unroll Loop Four Times (straightforward way)

1 cycle stall

2 cycles stall

```
1  Loop:LD    F0,0(R1)
2       ADDD  F4,F0,F2
3       SD    0(R1),F4     ;drop SUBI & BNEZ
4       LD    F6,-8(R1)
5       ADDD  F8,F6,F2
6       SD    -8(R1),F8    ;drop SUBI & BNEZ
7       LD    F10,-16(R1)
8       ADDD  F12,F10,F2
9       SD    -16(R1),F12  ;drop SUBI & BNEZ
10      LD    F14,-24(R1)
11      ADDD  F16,F14,F2
12      SD    -24(R1),F16
13      SUBI  R1,R1,#32    ;alter to 4*8
14      BNEZ  R1,LOOP
15      NOP
```

Rewrite loop to minimize stalls?

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration

Assumes R1 is multiple of 4

Unrolled Loop That Minimizes Stalls

```
1  Loop:LD      F0,0(R1)
2       LD      F6,-8(R1)
3       LD      F10,-16(R1)
4       LD      F14,-24(R1)
5       ADDD    F4,F0,F2
6       ADDD    F8,F6,F2
7       ADDD    F12,F10,F2
8       ADDD    F16,F14,F2
9       SD      0(R1),F4
10      SD      -8(R1),F8
11      SD      -16(R1),F12
12      SUBI    R1,R1,#32
13      BNEZ    R1,LOOP
14      SD        (R1),F16
```

stall

15 clock cycles per 4 iterations

Unrolled Loop That Minimizes Stalls

```
1  Loop:LD     F0,0(R1)
2       LD     F6,-8(R1)
3       LD     F10,-16(R1)
4       LD     F14,-24(R1)
5       ADDD   F4,F0,F2
6       ADDD   F8,F6,F2
7       ADDD   F12,F10,F2
8       ADDD   F16,F14,F2
9       SD     0(R1),F4
10      SD     -8(R1),F8
11      SUBI   R1,R1,#32
12      SD     16(R1),F12
13      BNEZ   R1,LOOP
14      SD     8(R1),F16     ; 8-32 = -24
```

14 clock cycles, or 3.5 per iteration

Compiler Perspectives on Code Movement

- Compiler try to schedule to avoid hazards
- Concerns about dependencies in a program
- True Data dependencies:
  - Instruction i produces a result used by instruction j, or
  - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.
- If dependent, they cannot be executed in parallel
- Easy to determine for registers (fixed names)
- Hard for memory:
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6)=20(R6)?

Summary of Loop Unrolling

- Determine if it is legal to move the instructions and adjust the offset
- Determine the unrolling loop would be useful by finding if the loop iterations were independent
- Use different registers to avoid unnecessary constraints.
- Eliminate the extra tests and branches
- Determine that the loads and stores in the unrolling loop are independent and can be interchanged (analyze the memory addresses and find that they do not refer to the same address)
- Schedule the code, preserving any dependences needed

Dependency

- Data dependence (true dependence)

- Name dependence
  - Anti-dependence (corresponds to WAR hazard)
  - Output dependence (corresponds to WAW hazard)

- Control dependence

Data Dependences

- Instruction i produces a result used by instruction j, or

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

Where are the data dependencies?

```
Loop:  LD       F0,0(R1)
       ADDD     F4,F0,F2
       SD       0(R1),F4
       SUBI     R1,R1,8
       BNEZ     R1, Loop
```

data dependencies

Where are the data dependencies

```
Loop:  LD      F0,0(R1)
       ADDD    F4,F0,F2
       SD      0(R1),F4
       SUBI    R1,R1,8
       BNEZ    R1, Loop
```

Importance of Data Dependencies

- Indicate the possibility of a hazard
- Determine the order in which result must be calculated
- Set an upper bound on how much parallelism can possibly be exploited

Overcoming the limitations of data dependence

- **Maintaining the dependence but avoiding a hazard**

- **Elimination a dependence by transforming the code**

Name Dependence

- Two instructions use the same name (register or memory location), but do not exchange data

- Anti-dependence (WAR)
  - Instruction j writes a register or memory location that instruction i read from
  - Instruction i is executed first

- Output dependence (WAW)
  - Instruction i and instruction j write the same register or memory location
  - Ordering between instructions must be preserved

Where are the name dependencies?

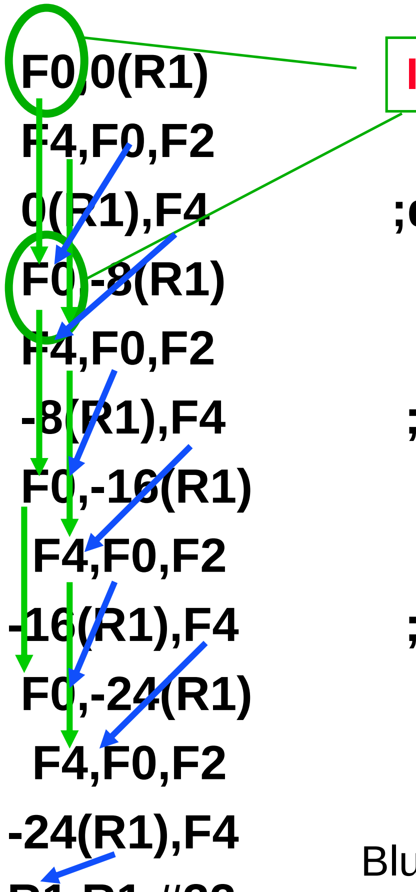
- Registers
- Memories

```
1 Loop: LD     F0,0(R1)
2       ADDD   F4,F0,F2
3       SD     0(R1),F4       ;drop SUBI and BNEZ
4       LD     F0,-8(R1)
5       ADDD   F4,F0,F2
6       SD     -8(R1),F4      ;drop SUBI and BNEZ
7       LD     F0,-16(R1)
8       ADDD   F4,F0,F2
9       SD     -16(R1),F4     ;drop SUBI and BNEZ
10      LD     F0,-24(R1)
11      ADDD   F4,F0,F2
12      SD     -24(R1),F4
13      SUBI   R1,R1,#32
14      BNEZ   R1,Loop
15      NOP
```

name dependencies (registers)

name dependencies (memory)

Where are the name dependencies (registers)



```
1 Loop: LD      F0,0(R1)
2       ADDD    F4,F0,F2
3       SD      0(R1),F4       ;drop SUBI and BNEZ
4       LD      F0,-8(R1)
5       ADDD    F4,F0,F2
6       SD      -8(R1),F4      ;drop SUBI and BNEZ
7       LD      F0,-16(R1)
8       ADDD    F4,F0,F2
9       SD      -16(R1),F4     ;drop SUBI and BNEZ
10      LD      F0,-24(R1)
11      ADDD    F4,F0,F2
12      SD      -24(R1),F4
13      SUBI    R1,R1,#32
14      BNEZ    R1,Loop
15      NOP
```

If we want to move Load together…

Blue: write after read (WAR) : antidependence

Green: write after write (WAW)

Where are the name dependencies (memory)

```
1 Loop: LD      F0,0(R1)
2       ADDD    F4,F0,F2
3       SD      0(R1),F4          ;drop SUBI and BNEZ
4       LD      F0,-8(R1)
5       ADDD    F4,F0,F2
6       SD      -8(R1),F4          ;drop SUBI and BNEZ
7        LD     F0,-16(R1)
8        ADDD    F4,F0,F2
9       SD     -16(R1),F4          ;drop SUBI and BNEZ
10      LD      F0,-24(R1)
11       ADDD    F4,F0,F2
12      SD     -24(R1),F4
13      SUBI   R1,R1,#32
14      BNEZ   R1,Loop
15      NOP
```

Where are the true dependencies?

```
1 Loop: LD      F0,0(R1)
2       ADDD    F4,F0,F2
3       SD      0(R1),F4          ;drop SUBI and BNEZ
4       LD      F6,-8(R1)
5       ADDD    F8,F6,F2
6       SD      -8(R1),F8         ;drop SUBI and BNEZ
7       LD      F10,-16(R1)
8       ADDD    F12,F10,F2
9       SD      -16(R1),F12       ;drop SUBI and BNEZ
10      LD      F14,-24(R1)
11      ADDD    F16,F14,F2
12      SD      -24(R1),F16
13      SUBI    R1,R1,#32
14      BNEZ    R1,Loop
15      NOP
```

True dependencies

Register Renaming

```
1 Loop: LD      F0,0(R1)
2       ADDD    F4,F0,F2
3       SD      0(R1),F4          ;drop SUBI and BNEZ
4       LD      F6,-8(R1)
5       ADDD    F8,F6,F2
6       SD      -8(R1),F8          ;drop SUBI and BNEZ
7        LD     F10,-16(R1)
8        ADDD    F12,F10,F2
9       SD      -16(R1),F12        ;drop SUBI and BNEZ
10      LD      F14,-24(R1)
11       ADDD    F16,F14,F2
12      SD      -24(R1),F16
13      SUBI    R1,R1,#32
14      BNEZ    R1,Loop
15      NOP
```

True dependencies

Compiler Perspectives on Code Movement

- Again Name Dependencies are hard for memory accesses
  - Does 100(R4)=20(R6)?
  - From different loop iterations, does 20(R6)=20(R6)?

- Our example required compiler to know that if R1 doesn't change, then

  0(R1) ≠ -8(R1) ≠ -16(R1) ≠ -24(R1)

- There were no dependencies between some loads and stores, so they could be moved/exchanged

Minimizes Stalls

```
1  Loop:LD     F0,0(R1)
2       LD     F6,-8(R1)
3       LD     F10,-16(R1)
4       LD     F14,-24(R1)
5       ADDD   F4,F0,F2
6       ADDD   F8,F6,F2
7       ADDD   F12,F10,F2
8       ADDD   F16,F14,F2
9       SD     0(R1),F4
10      SD     -8(R1),F8
11      SUBI   R1,R1,#32
12      SD     16(R1),F12
13      BNEZ   R1,LOOP
14      SD     8(R1),F16     ; 8-32 = -24    (delay slot)
```

Control Dependence

Example:

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

– **S1 is control dependent on p1**
– **S2 is control dependent on p2 but not on p1**

Compiler Perspectives on Code Movement

Two constraints on 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

- An instruction that is not control dependent on a branch cannot be moved to after the branch so that its execution is controlled by the branch

Control Dependency

- Two properties critical to program correctness, normally preserved by control dependency
  - Exception Behavior
  - Data Flow
- Exception Behavior
  - Any change in the ordering of execution must not change how exceptions are raised in the program
  - Example:

```
                BEQZ R2, L1
                LW     R1, 0(R2)
      L1:
```

  - No data dependency
  - If we move the Load before the Branch, may cause a memory protection exception

Control Dependency

- Data Flow
  - Branch makes the data flow dynamic.
  - Example:

```
            ADD    R1,R2,R3
            BEQZ  R4, L
            SUB    R1, R5,R6
  L:        OR     R7, R1, R8
```

  - Value of R1 in OR instruction depends on whether the branch is taken or not
  - I.e. OR is data dependent on both ADD and SUB

- Cannot change the data flow illegally.

Violating the control dependence may not affect the data flow

```
            ADD   R1, R2, R3
            BEQZ R12, skipnext
            SUB   R4,R5,R6
            ADD   R5,R4,R9
Skipnext:   OR    R7,R8,R9
```

Liveness: whether a value will be used by an upcoming instruction

- **If R4 unused after label skipnext, R4 is *dead* *(rather than live)***
  - **In this case, we could move SUB before the branch.**
  - **This is called "Compiler-based Speculation."**

Program Correctness

- Preserving Control Dependences
- When Safe to Unroll Loops?
  - Example: Where are data dependencies?

```
for (i=1;i<100;i=i+1)
    A[i+1]=A[i]+C[i]; //S1
    B[i+1]=B[i]+A[i+1]; //S2
end
```

- S2 uses the value A[i+1] computed by S1 in the same iteration.
- S1 uses a value computed by S1 in an earlier iteration: called Loop-carried dependence. (The same is true of S2 for B[i] and B[i+1])
- Iterations are dependent and cannot be executed in parallel.
- Note that in our prior example, each iteration was distinct and independent.

Loop carried dependence

Recurrence

```
For (i=2;i<=100;i=i+1) {
        Y[i]=Y[i-1]+Y[i];
}
```

Dependence distance of 1

```
For (i=6;i<=100;i=i+1){
        Y[i]=Y[i-5]+Y[i];
}
```

Dependence distance of 5

The larger the distance, the more potential parallelism can be obtained by unrolling the loop.

Another example on Loop-carried dependence (1/3)

- **Example:**

```
For (i=1;i<=100;i=i+1){
   A[i]=A[i]+B[i];          //S1
   B[i+1]=C[i]+D[i];        //S2
}
```

- **Is there a Loop-carried dependence ?**
- **Is the dependence circular?**
- **Can the loop be made parallel?**

Another example on Loop-carried dependence (2/3)

- **It is possible to have a Loop-carried dependence that does not prevent parallelism**

- **Example:**

```
  For (i=1;i<=100;i=i+1){
     A[i]=A[i]+B[i];         //S1
     B[i+1]=C[i]+D[i];     //S2
  }
```

- **There is a Loop-carried dependence between S1 and S2 (S1 uses the value assigned in the previous iteration by statement S2)**
- **Unlike the earlier example, the dependence is not circular. (S1 depends on S2, but S2 does not depend on S1. Neither statement depends on itself.)**
- **The loop can be made parallel.**

Another example on Loop-carried dependence (3/3): Rewrite the loop

```
For (i=1;i<=100;i=i+1){
    A[i]=A[i]+B[i];          //S1
    B[i+1]=C[i]+D[i];        //S2
}
```

- transforming the code

- After Rewriting:

```
A[1]=A[1]+B[1];
For (i=1;i<=99;i=i+1){
        B[i+1]=C[i]+D[i];
        A[i+1]=A[i+1]+B[i+1];
}
B[101]=C[100]+D[100];
```

- Dependence between the two statements is no longer loop-carried

- Iterations of the loop may be overlapped, provided the statements in each iteration are in order

3 Limits to Loop Unrolling

1. Decrease in amount of overhead amortized with each extra unrolling
   - Amdahl's Law
2. Growth in code size
   - For larger loops, concern it increases the instruction cache miss rate
3. Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling
   - If not be possible to allocate all live values to registers, may lose some or all of its advantage

- Loop unrolling reduces impact of branches on pipeline; another way is branch prediction

Review: Loop Unrolling Decisions

- Requires understanding
  - how one instruction depends on another
  - how the instructions can be changed or reordered given the dependences:

1. Determine loop unrolling useful by finding that loop iterations were independent

2. Use different registers to avoid unnecessary constraints forced by using same registers for different computations

Review: Loop Unrolling Decisions

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

4. Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent
   - Transformation requires analyzing memory addresses and finding that they do not refer to the same address

5. Schedule the code, preserving any dependences needed to yield the same result as the original code

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- **SuperScalar**
  - Varying number of instructions per cycle (1~8)
  - Scheduled by compiler or hardware (Tomasulo)

- **Very Long Instruction Words (VLIW)**
  - Fixed number of instructions (4~16)
  - Scheduled by compiler
  - Put operations into wide templates

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- **Superscalar DLX: 2 instructions, 1 FP & 1 anything else**
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - More ports for FP registers to do FP load & FP op in a pair

Type	*PipeStages*						
Int. instruction	IF	ID	EX	MEM	WB		
FP instruction	IF	ID	EX	MEM	WB		
Int. instruction		IF	ID	EX	MEM	WB	
FP instruction		IF	ID	EX	MEM	WB	
Int. instruction			IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB

- **1 cycle load delay expands to 3 instructions in Superscalar (SS)**
- **Would need additional register ports**
- **Would need pipelined or multiple floating-point units**

Loop Unrolling in Superscalar

```
Loop:  LD      F0,0(R1)
       ADDD    F4,F0,F2
       SD      0(R1),F4
       SUBI    R1,R1,8
       BNEZ    R1, Loop
```

	Integer instruction	*FP instruction*	*Clock cycle*
Loop:	LD F0,0(R1)		1
	LD F6,-8(R1)		2
	LD F10,-16(R1)	ADDD F4,F0,F2	3
	LD F14,-24(R1)	ADDD F8,F6,F2	4
	LD F18,-32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD -8(R1),F8	ADDD F20,F18,F2	7
	SD -16(R1),F12		8
	SD -24(R1),F16		9
	SUBI R1,R1,#40		10
	BNEZ R1,LOOP		11
	SD -32(R1),F20		12

- Unrolled 5 times to avoid delays
- 12 clocks, or 2.4 clocks per iteration

VLIW: Very Large Instruction Word

- Each "instruction" has explicit coding for multiple operations
- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - » 16 to 24 bits per field
  - Need compiling technique that schedules across several branches (Trace scheduling)

Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2			6
SD -16(R1),F12	SD -24(R1),F16				7
SD -32(R1),F20	SD -40(R1),F24			SUBI R1,R1,#48	8
SD -0(R1),F28				BNEZ R1,LOOP	9

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration

Average: 2.5 ops per clock, 50% efficiency

Note: Need more registers in VLIW

Limitations in Multiple-Issue Processors

- **Inherent limitations of available ILP in programs**
  - **To fill the issue slots and keep the functional units busy**
  - **Functional units that are pipelined or with multi-cycle latency require a larger number of operations that can be executed in parallel**
  - **# independent operations roughly equals to the average pipeline depth times the number of functional units**

- **Difficulties in building the underlying hardware**
  - **Hardware cost:**
    - » **Multiple floating-point and integer functional units**
    - » **Large increase in the memory bandwidth and Register-file bandwidth**

Static Scheduling and Dynamic Scheduling

- Compiler techniques for scheduling the instructions: Static Scheduling

- Hardware schemes: Dynamic Scheduling

- Why hardware at run time?
  - Works when we don’t know real dependence at compile time
  - Keep compiler simpler
  - Compiled code for one machine runs well on another

- Allow instructions behind stall to proceed
  - Enables out-of-order execution -> out-of-order completion