Instruction Level Parallelism

or

Declaration of Independence

What is ILP?

- The characteristic of a program that certain instructions are independent, and can potentially be executed in parallel.
- Any mechanism that creates, identifies, or exploits the independence of instructions, allowing them to be executed in parallel.

Where do we find ILP?

- In basic blocks?
  - 15-20% of (dynamic) instructions are branches in typical code
- Across basic blocks?
  - how?

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

A sample loop

```
Loop:  LD    F0,0(R1)     ;F0=vector element
       ADDD  F4,F0,F2     ;add scalar in F2
       SD    0(R1),F4     ;store result
       SUBI  R1,R1,8      ;decrement pointer 8B (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**

Data Hazards

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

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

- *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        BNEZ  R1,Loop     ;branch R1!=zero
9        stall             ;delayed branch slot
```

- *Rewrite code to minimize stalls?*

Revised FP Loop Minimizing Stalls

```
1 Loop:  LD    F0,0(R1)
2        stall
3        ADDD  F4,F0,F2
4        SUBI  R1,R1,8
5        BNEZ  R1,Loop    ;delayed branch
6        SD    8(R1),F4   ;altered when move past SUBI
```

Unroll loop 4 times code to make faster?

Unroll Loop Four Times

```
Loop:   LD      F0,0(R1)
        ADDD    F4,F0,F2
        SD      0(R1),F4        ;drop BNEZ
        SUBI    R1,R1,#8
        LD      F6,0(R1)
        ADDD    F8,F6,F2
        SD      0(R1),F8        ;drop BNEZ
        SUBI    R1,R1,#8
        LD      F10,0(R1)
        ADDD    F12,F10,F2
        SD      0(R1),F12       ;drop BNEZ
        SUBI    R1,R1,#8
        LD      F14,0(R1)
        ADDD    F16,F14,F2
        SD      0(R1),F16
        SUBI    R1,R1,#8
        BNEZ    R1,LOOP
        NOP
```

Unroll Loop Four Times

```
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      8(R1),F16      ; 8-32 = -24
```

14 clock cycles, or 3.5 per iteration

- What assumptions made when moved code?
  - OK to move store past SUBI even though changes register
  - OK to move loads before stores: get right data?
  - When is it safe for compiler to do such changes?

Compiler Perspectives on Code Movement

- Definitions: compiler concerned about dependencies in program, whether or not a HW hazard depends on a given pipeline
- (True) Data dependencies (RAW if a hazard for HW)
  - 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.
- Easy to determine for registers (fixed names)
- Hard for memory:
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?

Compiler Perspectives on Code Movement

- Definitions: compiler concerned about *dependencies* in program, whether or not a HW hazard depends on a given pipeline
- (*True*) Data dependencies (RAW if a hazard for HW)
  - 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.
- Easy to determine for registers (fixed names)
- Hard for memory:
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?

Compiler Perspectives on Code Movement

- Another kind of dependence called *name dependence*: two instructions use same name but don't exchange data
- *Antidependence* (WAR if a hazard for HW)
  - Instruction j writes a register or memory location that instruction i reads from and instruction i is executed first
- *Output dependence* (WAW if a hazard for HW)
  - Instruction i and instruction j write the same register or memory location; ordering between instructions must be preserved.

```
LD      F0,0(R1)
ADDD    F4,F0,F2
SD      0(R1),F4
LD      F0,-8(R1)
ADDD    F4,F0,F2
```

Compiler Perspectives on Code Movement

- Again 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 by each other

---

Compiler Perspectives on Code Movement

- Final kind of dependence called *control dependence*
- Example

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

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

Compiler Perspectives on Code Movement

- Two (obvious) constraints on 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 to *after* the branch so that its execution is controlled by the branch.
- Control dependencies relaxed to get parallelism; get same effect if preserve order of exceptions and data flow

When Safe to Unroll Loop?

- Example: Where are data dependencies?
  (A,B,C distinct & nonoverlapping)

```
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 */
```

  1. S2 uses the value, A[i+1], computed by S1 in the same iteration.

  2. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].
  This is a "*loop-carried dependence*": between iterations
- Implies that iterations are dependent, and can't be executed in parallel
- Not the case for our example; each iteration was distinct

Key Points

- You can find, create, and exploit Instruction Level Parallelism in SW or HW
- Loop level parallelism is usually easiest to see
- SW parallelism dependencies defined for program, and become hazards if HW cannot resolve
- SW dependencies/compiler sophistication determine if compiler can/should unroll loops

Code Motion

- Can be done in SW or HW
- Why SW?
- Why HW?

HW Schemes: Instruction Parallelism

- Why in HW at run time?
  - Works when can't know dependence at run time
  - Compiler simpler
  - Code for one machine runs well on another
- Key idea: Allow instructions behind stall to proceed

```
DIVD  F0,F2,F4
ADDD  F10,F0,F8
SUBD  F12,F8,F14
```

  - Enables out-of-order execution => out-of-order completion
  - ID stage checked both for structural & data dependencies

Out-of-order Issue/Dynamic Scheduling

- Problem -- need to get stalled instructions out of the ID stage, so that subsequent instructions can begin execution.
- Must separate detection of structural hazards from detection of data hazards
- Must split ID operation into two:
  - Issue (decode, check for structural hazards)
  - Read operands (read operands when NO DATA HAZARDS)
- i.e., must be able to issue even when a data hazard exists
- instructions issue in-order, but proceed to EX out-of-order

---

Dynamic Scheduling by hand

DIVD F0,F2,F4 (10 cycles)
ADDD F10, F0, F8 (4 cycles)
SUBD F12, F8, F14
ADDD F10,F2,F3
MULTD F13,F12,F2 (6 cycles)
ADDD F4,F1,F3
ADDD F5,F4,F0

CDC 6600 Scoreboard

- Enables dynamic scheduling
- Allows instructions to proceed when dependencies are satisfied

---

Scoreboard Implications

- Out-of-order completion => WAR, WAW hazards?
- Solutions for WAR
  - Queue both the operation and copies of its operands
  - Read registers only during Read Operands stage
- For WAW, must detect hazard: stall until other completes

DIVD F0,F2,F4
ADDD F10,F0,F8
SUBD F8,F8,F14

Scoreboard Implications, cont.

- Need to have multiple instructions in execution phase => multiple execution units or pipelined execution units
- Scoreboard keeps track of dependencies, state or operations
- Scoreboard replaces ID, EX, WB with 4 stages

Four Stages of Scoreboard Control

1. **Issue**—decode instructions & check for structural hazards (ID1)

   If a functional unit for the instruction is free and no other active instruction has the same destination register (WAW), the scoreboard issues the instruction to the functional unit and updates its internal data structure. If a structural or WAW hazard exists, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared.

2. **Read operands**—wait until no data hazards, then read operands (ID2)

   A source operand is available if no earlier issued active instruction is going to write it, or if the register containing the operand is being written by a currently active functional unit. When the source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution. The scoreboard resolves RAW hazards dynamically in this step, and instructions may be sent into execution out of order.

Four Stages of Scoreboard Control

3. **Execution**—operate on operands (EX)

   The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution.

4. **Write result**—finish execution (WB)

   Once the scoreboard is aware that the functional unit has completed execution, the scoreboard checks for WAR hazards. If none, it writes results. If WAR, then it stalls the instruction.

   Example:

```
DIVD    F0,F2,F4
ADDD    F10,F0,F8
SUBD    F8,F8,F14
```

   CDC 6600 scoreboard would stall SUBD until ADDD reads operands

---

Three Parts of the Scoreboard

1. **Instruction status**—which of 4 steps the instruction is in

2. **Functional unit status**—Indicates the state of the functional unit (FU). 9 fields for each functional unit

   Busy—Indicates whether the unit is busy or not

   Op—Operation to perform in the unit (e.g., + or –)

   Fi—Destination register

   Fj, Fk—Source-register numbers

   Qj, Qk—Functional units producing source registers Fj, Fk

   Rj, Rk—Flags indicating when Fj, Fk are ready

3. **Register result status**—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions will write that register

Scoreboard Example

Instruction status

Instruction		j	k	Issue	Read operands	Execution complete	Write Result
LD	F6	34+	R2				
LD	F2	45+	R3				
MULT	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

				dest	S1	S2	FU for j	FU for k	Fj?	Fk?
Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
FU									

Scoreboard Example

Instruction status

Instruction		j	k	Issue	Read operands	Execution complete	Write Result
LD	F6	34+	R2				
LD	F2	45+	R3				
MULT	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

				dest	S1	S2	FU for j	FU for k	Fj?	Fk?
Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
FU									

Scoreboard Example

Instruction status

Instruction		j	k	Issue	Read operands	Execution complete	Write Result
LD	F6	34+	R2				
LD	F2	45+	R3				
MULT	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

				dest	S1	S2	FU for j	FU for k	Fj?	Fk?
Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
FU									

Scoreboard Example

Instruction status

Instruction		j	k	Issue	Read operands	Execution complete	Write Result
LD	F6	34+	R2				
LD	F2	45+	R3				
MULT	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

				dest	S1	S2	FU for j	FU for k	Fj?	Fk?
Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
FU									

Scoreboard Example

Instruction status

Instruction		j	k	Issue	Read operands	Execution complete	Write Result
LD	F6	34+	R2				
LD	F2	45+	R3				
MULT	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

				dest	S1	S2	FU for j	FU for k	Fj?	Fk?
Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
	FU									

Scoreboard Example

Instruction status

Instruction		j	k	Issue	Read operands	Execution complete	Write Result
LD	F6	34+	R2				
LD	F2	45+	R3				
MULT	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

				dest	S1	S2	FU for j	FU for k	Fj?	Fk?
Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
	FU									