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
```


How do we expose ILP?

- by moving instructions around.
- How??
  - software
  - hardware

Exposing ILP in software

- instruction scheduling (changes ILP within a basic block)
- loop unrolling (allows ILP across iterations by putting instructions from multiple iterations in the same basic block)
- Others (trace scheduling, software pipelining)

A sample loop

```
Loop:   LD      F0,0(R1)      ;F0=array element, R1=X[]
        ADDD    F4,F0,F2      ;add scalar in F2
        SD      0(R1),F4      ;store result
        ADDI    R1,R1,8       ;increment pointer 8B (DW)
        SEQ     R3, R1, R2    ;R2 = &X[1001]
        BNEZ    R3,Loop       ;branch R3!=zero
        NOP                   ;delayed branch slot
```

Instruction producing result	***Instruction using result***	***Latency (stalls) in clock cycles***
FP ALU mult	**Another FP ALU op**	**6**
FP ALU mult	**Store double**	**5**
Load double	**FP ALU op**	**1**
Load double	**Store double**	**0**
Integer op	**Integer op**	**0**

Dependencies & Stalls

```
Loop:   LD      F0,0(R1)      ;F0=array element, R1=X[]
        ADDD    F4,F0,F2      ;add scalar in F2
        SD      0(R1),F4      ;store result
        ADDI    R1,R1,8       ;increment pointer 8B (DW)
        SEQ     R3, R1, R2    ;R2 = &X[1001]
        BNEZ    R3,Loop       ;branch R3!=zero
        NOP                   ;delayed branch slot
```

Instruction producing result	***Instruction using result***	***Latency in clock cycles***
FP ALU mult	**Store double**	**5**
Load double	**FP ALU op**	**1**

Instruction Scheduling

- stalls?

Loop Unrolling

- stalls?

Register Renaming

Compiler Perspectives on Code Movement

- Definitions: *dependencies* are a property of code, whether or not it is a HW *hazard* depends on the given pipeline.
- (*True*) Data dependencies (RAW)
  - 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 dependence)
  - Instruction j writes a register or memory location that instruction i reads from and instruction i is executed first
- *Output dependence* (WAW dependence)
  - 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
ADDI    R1,R1,8
SEQ     R3, R1, R2
BNEZ    R3,Loop
NOP
```

Compiler Perspectives on Code Movement

- Name Dependence Also 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)
```

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 (c1)
    I1;
if (c2)
    I2;
```

I1 is control dependent on c1 and I2 is control dependent on c2 but not on c1.

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

Code Motion

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

Key Points

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

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

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer									
	Mult1									
	Mult2									
	Add									
	Divide									

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

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer									
	Mult1									
	Mult2									
	Add									
	Divide									

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

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer									
	Mult1									
	Mult2									
	Add									
	Divide									

Register result status

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

CSE 240 Dean Tullsen

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

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer									
	Mult1									
	Mult2									
	Add									
	Divide									

Register result status

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

CSE 240 Dean Tullsen

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

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer									
	Mult1									
	Mult2									
	Add									
	Divide									

Register result status

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

CSE 240 Dean Tullsen

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

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for j Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer									
	Mult1									
	Mult2									
	Add									
	Divide									

Register result status

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