

Instruction level parallelism

- Potential overlap among instructions is called ILP
 - ⇒ Implies a lack of dependence between instructions.
 - All of the techniques in this chapter exploit parallelism among *instruction sequences*
- ILP techniques include
 - Static techniques (done by compiler)
 - Basic dynamic scheduling (done by hardware)
 - Additional “hidden” registers (done by hardware)
 - Branch prediction
 - Superscalar execution

How much can ILP help?

- Goal: execute as many instructions as possible in as little time as possible
 - Keep functional units busy with useful work
 - Execute instructions out of order
 - Use other techniques to do as much as possible at one time
 - Allow more functional units to be useful
- Limits to ILP
 - Processor: ILP limited by number and type of functional units
 - Program: interdependence of instructions can limit the instructions that can be done in parallel

ILP and CPI

- How is CPI calculated?
$$\text{CPI} = \text{ideal CPI} + \text{structural stalls} + \text{RAW stalls} + \text{WAW stalls} + \text{WAR stalls} + \text{control stalls}$$
- Previous chapter: reduce RAW & control stalls
- This chapter: reduce all components of the CPI equation
 - Additional reductions in RAW & control stalls
 - Reduce ideal CPI
 - Use more hardware to reduce or eliminate structural stalls

Techniques for reducing CPI via ILP

- Loop unrolling (reduces control stalls)
- Basic pipeline scheduling (reduces RAW stalls)
- Scoreboarding (reduces RAW stalls)
- Register renaming (reduces both WAR and WAW stalls)
- Dynamic branch prediction (reduces control stalls)
- Issuing multiple instructions per cycle (reduces ideal CPI)
- Compiler dependence analysis (reduces ideal CPI and data stalls)
- Software pipelining and trace scheduling (reduces ideal CPI and data stalls)
- Speculation (execute “possible” instructions, reducing data & control stalls)
- Dynamic memory disambiguation (reduce RAW memory stalls)

Where does ILP come from?

- ILP comes from basic blocks
 - Block of code with no branches into the code except at the start and no branches out of the code except at the end
 - Code inside the average basic block is quite small
 - Average dynamic branch frequency in integer programs $\approx 15\%$
 - About 6 to 7 instructions are executed between a pair of branches
 - Average = usual case?
 - Instructions in basic block depend on one another because they tend to operate on the same data in sequence
- ⇒ Exploit ILP across multiple basic blocks!

Loop-level parallelism

- Exploit parallelism among iterations of a loop
 - Iterations of a loop are often independent
 - Each iteration can overlap with any other iteration even though individual iterations have few (if any) overlappable instructions
- Techniques exist for exploiting the ILP in loops
 - Done statically by the compiler (loop unrolling)
 - Done dynamically by the CPU
 - Vector processors can run very quickly on simple loop operations

```
for (j = 0; j < 2000; j++) {
    dp[j] = x[j] * y[j] + z[j];
}
```

Pipeline scheduling

- Compiler tries to separate a dependent instruction from the source instruction so there's no data stalls
 - Compiler must have intimate knowledge of the internal hardware workings
 - Code optimized for one version of a processor may not be optimized on a future version of the processor...
- Assume the following latencies:

Instruction producing result	Instruction using result	Latency (clock cycles)
FP ALU operation	Another FP ALU op	3
FP ALU operation	Store double	2
Load double	FP ALU op	1
Load double	Store double	0 (using forwarding)

Pipeline scheduling example: before

- Compile the following code:


```
for (j = 0; j < 2000; j++)
    x[j] = x[j] + c;
```

 - Assume R1 holds the address of x[1999]
 - Assume F2 has the scalar value c
- Unscheduled code has many stalls!
 - 5 cycles of useful work
 - 5 cycles of stalls!
 - Total = 10 cycles per iteration

```
Loop:
LD    F0,0(R1) ; F0=array elem
ADD  F4,F0,F2 ; add scalar
SD   0(R1),F4 ; store result
SUBI R1,R1,#8 ; pointer--
BNEZ R1,Loop ; repeat loop
```

```
Loop:
LD    F0,0(R1) ; F0=array elem
STALL
ADD  F4,F0,F2 ; add scalar
STALL
STALL
SD   0(R1),F4 ; store result
SUBI R1,R1,#8 ; pointer--
STALL
BNEZ R1,Loop ; repeat loop
STALL
```

Pipeline scheduling: after

- Reschedule code to reduce stalls
 - Move SUBI earlier
 - Move SD later (and fix address)
- Still one stall
 - SD stalls 1 cycle waiting for ADDD result
- Reduced total time from 10 -> 6
 - Still only 3 cycles of work!
 - 2 instructions & 1 stall overhead
- Goal: get more “useful” operations per loop overhead
 => Replicate the loop body multiple times and adjust loop control

```

Loop:
LD  F0,0(R1) ; F0=array elem
ADDD F4,F0,F2 ; add scalar
SD  0(R1),F4 ; store result
SUBI R1,R1,#8 ; pointer--
BNEZ R1,Loop ; repeat loop

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

Loop unrolling

- Loop unrolling => create multiple copies of the loop body
 - Improves scheduling by
 - Eliminating branches (control hazards cost time!)
 - Allowing instructions from multiple iterations to be interleaved, exposing more parallelism
 - Allows CPU to amortize loop overhead across several loop iterations
 - Comparison at end of loop
 - Pointer / index increments
- Loop unrolling increases register usage
 - Better utilization of a scarce resource
 - More chance for cycles between uses of a register to be filled

Loop unrolling: example

- Unroll & schedule code from previous example
 - Unroll 4 times
 - Use displacement addressing mode to increment index once per “macro” loop
 - Assume $R1 \text{ MOD } 32 == 0$
- Code has no stalls!
 - 14 clock cycles for 4 elements
=> 3.5 clock cycles / element
 - Speedup of $6/3.5 = 1.7x$
 - Using different registers => avoid *false dependencies*
 - Reordering code eliminates stalls!

```

loop:
LD  F0,0(R1)
LD  F6,-8(R1)
LD  F10,-16(R1)
LD  F14,-24(R1)
ADDD F4,F0,F2
ADDD F8,F6,F2
ADDD F12,F10,F2
ADDD F16,F14,F2
SD  0(R1),F4
SD  -8(R1),F8
SUBI R1,R1,#32
SD  16(R1),F12
BNEZ R1,loop
SD  8(R1),F16
    
```

Loop unrolling: details

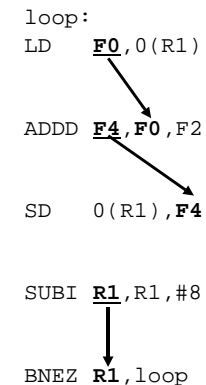
- What if loop index in example isn't a multiple of 32?
 - Real code: don't always know upper bound on loop
 - Real code: don't know how many times the loop will be executed!
- Solution: assume loop unrolled k times and iterated n times
 - First code body contains original code, and executes $n \text{ MOD } k$ times
 - Second code contains unrolled body and executes $\lfloor n/k \rfloor$ times
 - Saves time if the number of iterations was large
 - Another solution: jump into the middle of the code (if possible)
- Loop unrolling is easy to recognize
 - Not trivial for a compiler to perform these optimizations!
 - Compilers can, however, do scheduling *very* well

Dependencies: the basics

- Dependency \Rightarrow instruction B uses a result of instruction A
 - Dependencies are a property of programs, not of CPUs and pipelines.
 - Dependence between two instructions will always exist unless the program is changed
- Presence of a dependence indicates the *potential* for a hazard
 - Actual hazard and the length of any stall is a property of the pipeline
 - Goal is to eliminate stalls, not dependencies!
- Three types of dependencies
 - Data
 - Name
 - Control

Data dependence

- Instruction j is dependent on i if i produces a result used by j
- Dependence is transitive
 - j is dependent upon i and k is dependent upon j , $\Rightarrow k$ is dependent on i
- Dependence chains can be arbitrarily long!
- A compiler scheduling instructions cannot move j before i if j depends upon i



Overcoming data dependencies

- Data dependencies
 - Indicate the possibility of a hazard
 - Determine the order in which results must be generate
 - Place an upper bound on the amount of ILP available
- Data dependencies can be overcome in two ways
 - Keeping the dependence but avoiding a hazard
 - Eliminating the dependence by transforming the code
- Scheduling is the primary way to **avoid hazards** without altering dependencies
 - See previous example with LD, ADDD and SD
 - Code scheduled to avoid the hazard, but the dependence remained in the code

Eliminating data dependencies

- It's possible to eliminate data dependencies
 - Eliminate instructions!
 - Loop unrolling: eliminate branches and index updates
- Compiler removes dependence by eliminating instructions
 - BNEZ instructions dropped
 - Eliminate SUBI instructions & fold computation into offset

```

Loop:
LD   F0, 0(R1) ; F0=array elem
ADDD F4, F0, F2 ; add scalar
SD   0(R1), F4 ; store result
SUBI R1, R1, #8 ; pointer--
BNEZ R1, Loop ; repeat loop
LD   F6, 0(R1) ; F0=array elem
ADDD F8, F6, F2 ; add scalar
SD   0(R1), F8 ; store result
SUBI R1, R1, #8 ; pointer--
BNEZ R1, Loop ; repeat loop
LD   F10, 0(R1) ; F0=array elem
ADDD F12, F10, F2 ; add scalar
SD   0(R1), F12 ; store result
SUBI R1, R1, #8 ; pointer--
BNEZ R1, Loop ; repeat loop
    
```

More on data dependencies

- Eliminating data dependencies requires a fair amount of analysis => done by the compiler
- Avoiding hazards through scheduling can be done in hardware or software or both
- What about data dependence through a memory location?
 - Registers are easy to figure out at compile time
 - Memory dependences may not be known until runtime => much more difficult to deal with!
 - Example: 100(R4) and 20(R6) may refer to the same memory location
 - Not known until runtime, though!
 - Explore hardware and software techniques that detect data dependencies that involve memory locations (later...)



Name dependencies

- Two instructions use the same register or memory location, but there's an intervening write to it
 - These are **NOT** data dependencies because no information is passed between the two instructions
 - The instructions could be executed out of order or in parallel if the CPU renamed the register or memory location involved
- Register renaming can either be done by
 - Compiler, as in earlier loop unrolling
 - CPU (dynamic register renaming)
- In example
 - Second LD replaces value in F0
 - Second ADDD replaces value in F4

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



Unrolled loop before register renaming



Control dependencies

- A control dependency determines the ordering of the instructions with respect to branch instructions
 - If an instruction depends on the outcome of an earlier branch then it is only executed on one of the two forks
 - This instruction is dependent on the preceding branch
- Example:

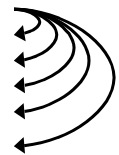

```
if (cond1)
    S1;
else
    S2;
```
- Obviously, we cannot:
 - Move S1 or S2 before the *if* statement
 - Move other instructions before the *if* stmt into “then” or “else”



Control dependencies

- In loop unrolling before removing branches:
 - Control hazards after each branch!
- Eliminated branches because we knew the outcome of each branch
 - Iterations divisible by 4
 - All “internal” branches taken (to top of loop)
 - Eliminated control dependencies!

Loop:
 LD F0,0(R1)
 ADDD F4,F0,F2
 SD 0(R1),F4
 SUBI R1,R1,#8
 BNEZ R1,Loop
 LD F6,0(R1)
 ADDD F8,F6,F2
 SD 0(R1),F8
 SUBI R1,R1,#8
 BNEZ R1,Loop
 ...



Preserving program correctness

- Preserving control dependence is NOT a critical property
 - Program can be rewritten to violate control dependence!
 - Program correctness is the critical property that must be preserved
- Violating control dependence may be OK if program correctness is preserved!
- Two properties critical to program correctness are
 - Preserving exception behavior: any changes in the ordering of instructions must NOT change how exceptions are raised
 - An instruction that should not have been executed can't cause an exception
 - Memory operations and floating point often cause problems like this
 - Preserving data flow



Preserving data flow

- Branches make the flow of information between instructions dynamic
 - Different values for particular registers depend on whether or not branches are taken
 - This information flow must be preserved!
- Data flow can be preserved by
 - CPU cancels instructions that were wrongly executed
 - Compiler cancels things out (add to cancel out a subtract that shouldn't have been executed)



Dealing with control dependencies

- Sometimes, violating control dependencies can't affect execution behavior or data flow

```
ADD R1,R2,R3
BEQZ R12,skipnext
SUB R4,R5,R6
ADD R5,R4,R9
skipnext:
OR R7,R8,R9
```
- Could move SUB before BEQZ if we knew
 - The SUB instruction could not generate an exception
 - If R4 were not 'live', i.e., used after the skipnext label
- This type of scheduling is called speculation: the compiler is betting that the branch will not be taken
 - Hardware can do this too...



Control dependencies: summary

- Control dependence is preserved by implementing control hazard detection
- Control hazard detection causes control stalls
- Control stalls can be avoided by:
 - Scheduling instructions in delay slots
 - Loop unrolling
 - Conditional execution
 - Speculation by both compiler and CPU
- We will cover the latter two shortly along with other dynamic methods for taking advantage of ILP

