Instruction-Level Parallelism (Part II)

(Chapter 3)

Outline

- Basic Compiler Technique for Exposing ILP
- Static Branch Prediction
- Static Multiple Issue: The VLIW Approach
- Multithreading
- Putting It All Together: The Intel Core I7 and ARM Cortex-A53 and A57

Basic Compiler Technique for Exposing ILP

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, 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.

Here we look at how the compiler can increase the amount of available ILP by unrolling loops.

Consider the example shown below.

```
For (i=1000;i>0;i=i-1)
     x[i]=x[i]+s;
```

We can see that this loop is parallel by noticing that the body of each iteration is independent.

Assume the basic 5-stage pipeline is used. The simple MIPS code for the loop is given by:

```
Loop:   L.D      F0,0(R1)
        ADD.D    F4,F0,F2
        S.D      F4,0(R1)
        ADDI     R1,R1,#-8
        BNE      R1,R2,Loop
```

Latencies of FP operations used in this chapter

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

Latency defined in this chapter: number of **intervening** clock cycles needed to avoid a stall.

For the integer operations, we note that
the branch instructions always introduce a stall.

Any instruction immediately following an integer ALU
operation, and using the results of the integer
ALU operation also introduces a stall.

Example:

Show how the loop would look, both scheduled and unscheduled, including any any stalls or idle clock cycles.

Sol:

We see that 6 and 10 clock cycles are required for each iteration with and without scheduling, respectively. For the loop with scheduling, we also find that the loop overhead is 3 clock cycles (BNE, ADDI and stall).

```
Loop: L.D      F0,0(R1)
      stall                   latency =1
      ADD.D    F4,F0,F2
      stall                   latency =2
      stall
      S.D      F4,0(R1)
      ADDI     R1,R1,#-8
      stall
      BNE      R1,R2,Loop
      stall
```

Loop without scheduling

```
Loop: L.D      F0,0(R1)
      ADDI     R1,R1,#-8      overhead
      ADD.D    F4,F0,F2
      stall                   overhead
      BNE      R1,R2,Loop     overhead
      S.D      F4,8(R1)
```

Loop with scheduling

A simple scheme for increasing the number of instructions relative to the branch and overhead instructions is **loop unrolling**. Unrolling simply replicates the loop body multiple times.

Example:

Show our loop unrolled so that there are four copies of the loop body, assuming R1 is initially a multiple of 32, which means that the number of loop iterations is a multiple of 4.

Sol:

```
Loop: L.D      F0,0(R1)         }
      ADD.D    F4,F0,F2         }  6 clock cycles
      S.D      F4,0(R1)         }
      L.D      F6,-8(R1)        }
      ADD.D    F8,F6,F2         }  6 clock cycles
      S.D      F8,-8(R1)        }
      L.D      F10,-16(R1)      }
      ADD.D    F12,F10,F2       }  6 clock cycles
      S.D      F12,-16(R1)      }
      L.D      F14,-24(R1)      }
      ADD.D    F16,F14,F2       }  6 clock cycles
      S.D      F16,-24(R1)      }
      ADDI     R1,R1,#-32          2 clock cycles
      BNE      R1,R2,Loop          2 clock cycles
                                ______________________
                                Total=28 clock cycles
```

Example:

Show the unrolled loop in the previous example after it has been scheduled for the pipeline.

Sol:

```
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)
      ADDI      R1,R1,#-32
      S.D       F12,16(R1); 16-32=-16
      BNE       R1,R2,Loop
      S.D       F16,8(R1); 8-32=-24
```

Total=14 cycles

Example:

Show how the process of optimizing the loop overhead by unrolling the loop actually eliminates data dependences.

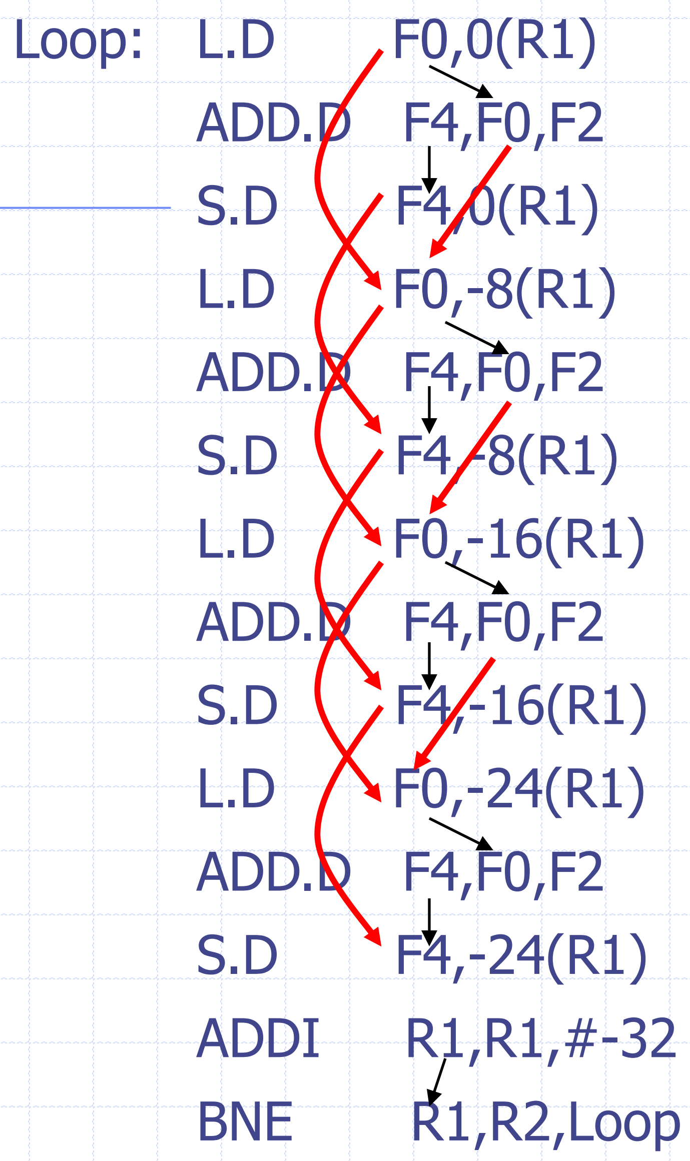
Sol:

```
Loop:  L.D      F0,0(R1)
       ADD.D    F4,F0,F2
       S.D      F4,0(R1)
       ADDI     R1,R1,#-8
       L.D      F6,0(R1)
       ADD.D    F8,F6,F2
       S.D      F8,0(R1)
       ADDI     R1,R1,#-8
       L.D      F10,0(R1)
       ADD.D    F12,F10,F2
       S.D      F12,0(R1)
       ADDI     R1,R1,#-8
       L.D      F14,0(R1)
       ADD.D    F16,F14,F2
       S.D      F16,0(R1)
       ADDI     R1,R1,#-8
       BNE      R1,R2,Loop
```

As the arrows show, the ADDI instructions form a dependent chain that involves the ADDI, L.D and S.D instructions. This chain forces the body to execute in order. **The compiler removes this dependence by symbolically computing the intermediate values of R1 and folding the computation into the offset of L.D and S.D instructions.** This transformation makes the three ADDI unnecessary, and compiler can remove them.

Example:

Unroll our example loop, eliminating the excess loop overhead, but using the same registers in each loop copy. Indicate both the data and name dependences within the body. Show how renaming eliminates name dependences that reduce parallelism.

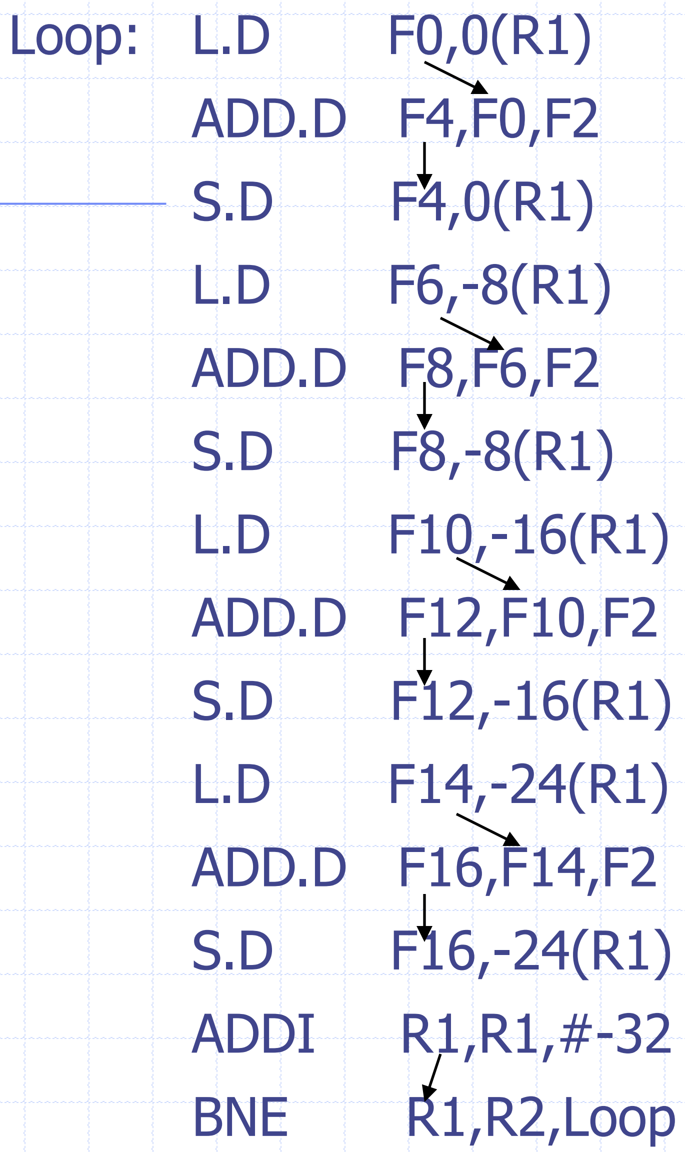


```
Loop:  L.D     F0,0(R1)
       ADD.D   F4,F0,F2
       S.D     F4,0(R1)
       L.D     F0,-8(R1)
       ADD.D   F4,F0,F2
       S.D     F4,-8(R1)
       L.D     F0,-16(R1)
       ADD.D   F4,F0,F2
       S.D     F4,-16(R1)
       L.D     F0,-24(R1)
       ADD.D   F4,F0,F2
       S.D     F4,-24(R1)
       ADDI    R1,R1,#-32
       BNE     R1,R2,Loop
```

Here is the loop unrolled but with the same registers in use for each copy.

$\longrightarrow$ Name dependence

$\longrightarrow$ Data dependence



```
Loop:   L.D      F0,0(R1)
        ADD.D    F4,F0,F2
        S.D      F4,0(R1)
        L.D      F6,-8(R1)
        ADD.D    F8,F6,F2
        S.D      F8,-8(R1)
        L.D      F10,-16(R1)
        ADD.D    F12,F10,F2
        S.D      F12,-16(R1)
        L.D      F14,-24(R1)
        ADD.D    F16,F14,F2
        S.D      F16,-24(R1)
        ADDI     R1,R1,#-32
        BNE      R1,R2,Loop
```

When the registers used for each copy of the loop body are renamed, only the true dependences within each body remain.

Using Loop Unrolling and Pipeline Scheduling with static Multiple Issue

Example:

Consider a simple two-issue, statically scheduled super scalar MIPS pipeline. Suppose the same example code segment is used. Unroll and schedule the loop in the code segment.

Sol: This unroll superscalar loop now runs in 12 clock cycles per iteration.

Clock Cycles	Integer Instructions	FP Instructions
1	Loop: L.D F0,0(R1)	
2	L.D F6,-8(R1)	2 clocks
3	L.D F10,-16(R1)	ADD.D **F4**,F0,F2
4	L.D F14,-24(R1)	ADD.D F8,F6,F2
5	L.D F18,-32(R1)	ADD.D F12,F10,F2
6	3 clocks S.D **F4**,0(R1)	ADD.D F16,F14,F2
7	S.D F8,-8(R1)	ADD.D F20,F18,F2
8	S.D F12,-16(R1)	
9	ADDI R1,R1,-40	
10	SD F16,16(R1)	
11	BNE R1,R2,Loop	
12	SD F20,8(R1)	

Static Branch Prediction

Compilers can use the static branch prediction for scheduling the code segments containing branch instructions.

The delayed branch is a scheme supporting the static branch prediction for reducing the penalty associated with the control hazard.

Conditional selection branches also employs the static branch for solving the data hazard.

Consider the following code segment:

```
        LD      R1,0(R2)
        SUB     R1,R1,R3          <----------  stall
        BEQZ    R1,L
        OR      R4,R5,R6
        ADD     R10,R4,R3
L:      ADD     R7,R8,R9
```

The dependence of SUB and BEQZ on LD means that a stall will be needed after the LD.

Branch was almost untaken

```
        LD      R1,0(R2)
        SUB     R1,R1,R3
        BEQZ    R1,L
        OR      R4,R5,R6
        ADD     R10,R4,R3
L:      ADD     R7,R8,R9
```

Branch was almost taken

Suppose we knew that the branch was almost taken, then we can move ADD R7,R8,R9 to the position after LD to increase the speed of the program.

Suppose the branch was almost untaken, we can move the OR instruction to the position after LD.

The following are the basic techniques for realizing the static branch prediction.

1. Always predict the branch as taken.
2. Select the backward-going branches to be taken and forward-going branches to be untaken.
3. Predict branches on the basis of profile information collected from earlier runs.

Static Multiple Issue: The VLIW Approach

VLIW processors issue a fixed number of instructions formatted as one large instruction.

In the VLIW, the compiler is required to do most of the work of finding and scheduling instructions for parallel execution. The hardware need not check explicitly for dependence.

An example of VLIW processor might have instructions that contain five operations, including one integer operation (which could also be a branch), two floating-point operations, and two memory references.

To keep the functional units busy, there must be enough parallelism in a code sequence to fill the available operation slots. The compiler might use local scheduling and/or global scheduling techniques for uncovering the parallelism.

Example:

Suppose we have a VLIW that could issue two memory references, two FP operations, and one integer operation or branch in every clock cycle. Show an unrolled version of the loop x[i]=x[i]+s for such a processor.

Sol:

Cloc k	Memory Reference 1	Memory Reference 2	FP Operations 1	FP Operations 2	Integer Operations
1	LD F0,0(R1)	LD F6,-8(R1)			
2	LD F10,-16(R1)	LD F14,-24(R1)			
3	LD F18,-32(R1)	LD F22,-40(R1)	ADD **F4**,F0,F2	ADD F8,F6,F2	
4	LD F26,-48(R1)		ADD F12,F10,F2	ADD F16,F14,F2	
5			ADD F20,F18,F2	ADD F24,F22,F2	
6	SD **F4**,0(R1)	SD F8,-8(R1)	ADD F28,F26,F2		
7	SD F12,-16(R1)	SD F16,-24(R1)			ADDI R1,R1,-56
8	SD F20,24(R1)	SD F24,16(R1)			
9	SD F28,8(R1)				BNE R1,R2,Loop

The technical problem of the original VLIW model are the increase in code size and the limitations of lockstep operation.

Two elements combine to increase code size. First, generating enough operations in a straight-line code fragment requires ambitiously unrolling loops. Second, whenever instructions are not full, the unused functional units translate to waste bits in the instruction encoding.

In the early VLIWs, a stall in any functional unit pipeline must cause the entire processor to stall, since all the functional units must be kept synchronized.

Multithreading: Exploiting Thread-Level Parallelism within a Processor

Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion.

To permit this sharing, the processor must duplicate the independent state of each thread.

For example: Register file, PC, and page table.

There are two main approaches to multithreading: **Fine-grained** multithreading and **coarse-grained** multithreading.

Fine-grained multithreading switches between threads on **each instruction**, causing the execution of multiple threads to be interleaved.

Advantage: Hide the throughput losses that arise from both short and long stalls.

Disadvantage: Slow down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

Coarse-grained multithreading switches threads only on costly stalls, such as L2 cache misses.

Advantage: Less likely to slow down the execution of an individual thread.

Disadvantage: Need start-up overhead when switching threads.

Simultaneous multithreading: converting thread-level parallelism (TLP) into instruction-level parallelism

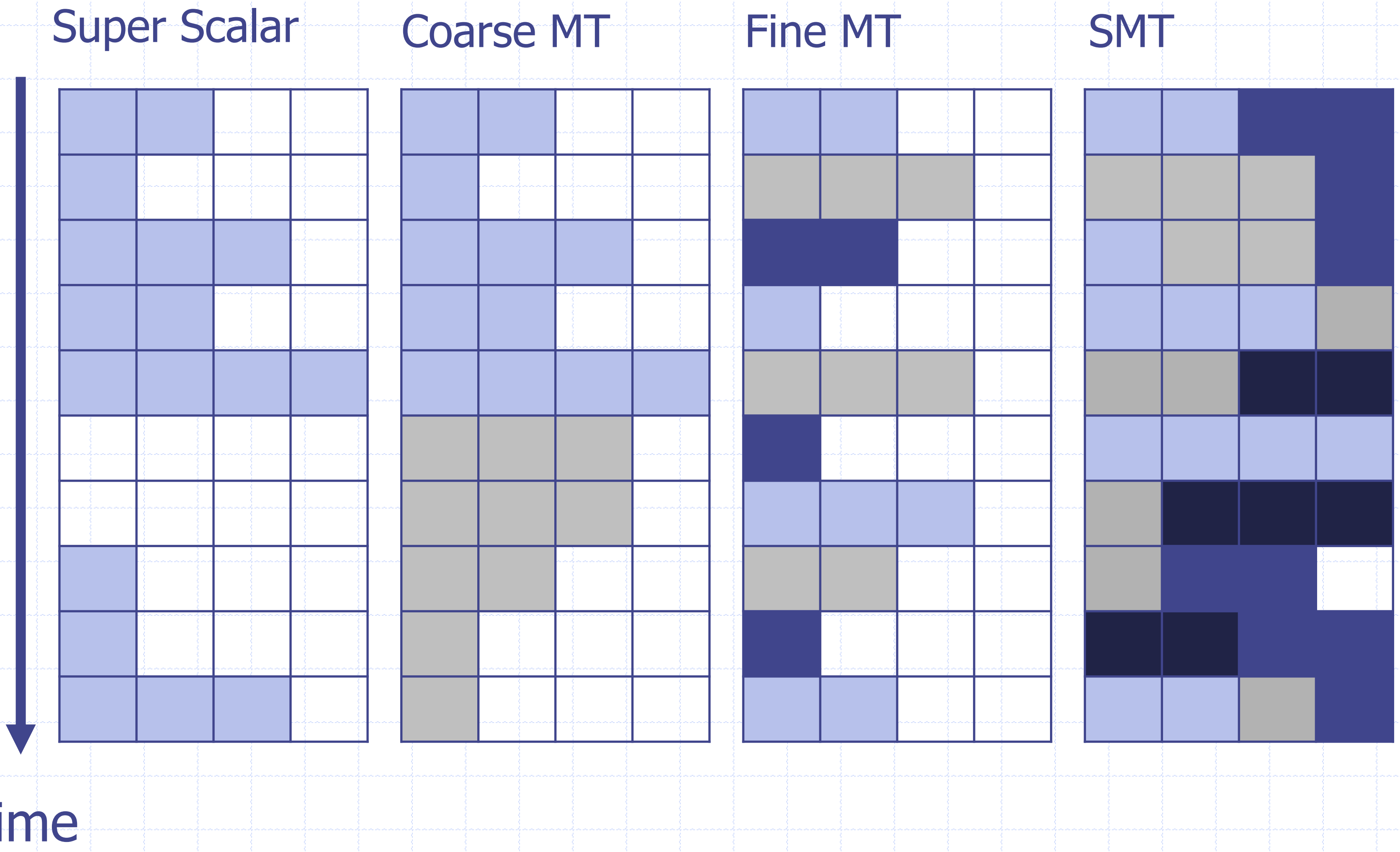
Simultaneous multithreading (SMT) is a variation on multithreading that uses the resources of a multiple-issue, dynamically scheduled processor to exploit TLP at the same time it exploits ILP.

Motivation: Modern multiple-issue processors often have more functional unit parallelism available than a single thread can effectively use.

In the SMT, TLP and ILP are exploited simultaneously, with **multiple threads** using the issue slots in a **single clock time**.

Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads.

An example showing the potential advantages of multithreading in general and SMT in particular for the processors exploiting the resources of a superscalar.



Because a dynamically scheduled superscalar processor is likely to have a deep pipeline, SMT will be unlikely to gain much in performance if it were coarse-grained.

Since SMT will likely make sense only in fine-grained implementation, the impact of fine-grained scheduling on single-thread performance should be considered. This effect can be minimized by having a preferred thread, which still permits multithreading to preserve some of its advantage with a smaller compromise in single-thread performance.

Putting It All Together: The Intel Core I7 and ARM Cortex-A53, A57, A72

- The ARM Cortex-A53, A57 and A72

The ARM Cortex-A53, A57 and A72 cores can be used as basis for tablets, or cell phones. They support floating point operations.

Cortex- A53: Raspberry PI 3, Qualcomm Snapdragon 41x, 42x, 43x, 61x, 62x (HTC Desire 820, Samsung Galaxy A7)

Cortex- A57: Qualcomm Snapdragon 810 (HTC One M9, Sony Xperia Z5)

Cortex- A72: Raspberry PI 4, Qualcomm Snapdragon 650 (Samsung Galaxy A9)

The A53 is a dual-issue, statically scheduled superscalar with dynamic issue detection, which allows the processor to issue one or two instructions per clock. The A53 pipeline contains 8 stages.

The A57 and A72 are triple-issue, speculative out-of-order superscalar pipeline. They have 14 stages for integer operations.

Intel Core I7

The I7 uses an aggressive out-of-order speculative microarchitecture.

The total pipeline depth is 14 stages.

Individual x86 instructions are translated into micro-ops. Micro-ops are simple RISC-V-like instructions that can be executed directly by the pipeline.

The I7 uses a 36-entry reservation station shared by six functional units. Up to six micro-ops may be dispatched to the functional units every clock cycle.