Lecture 15

Computer Architecture
(Spring 2020)

Instruction-Level Parallelism

Dr. Duo Liu (刘铎)
Office: Main Building 0626
Email: liuduo@cqu.edu.cn

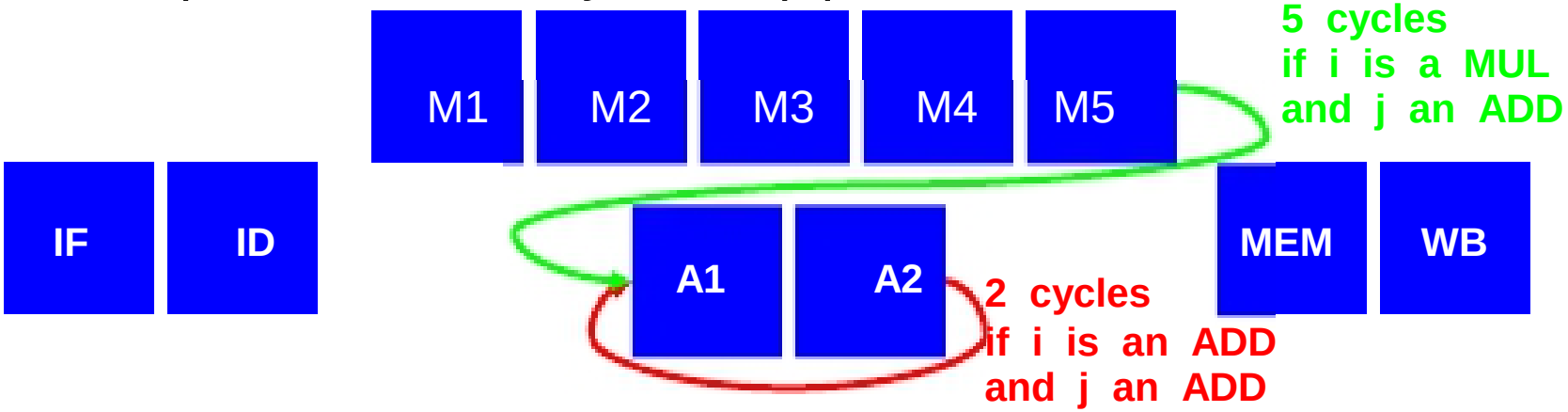
Compiler Technology to Improve ILP

- Loop Unrolling & Pipeline Static Scheduling
- Software Pipelining
- Compiler-Based Branch Prediction

- Applications
  - processors that use static issue
    - VLIW and VLIW-like
      - NXP Semiconductors (founded by Philips) Trimedia
      - Transmeta Crusoe
  - processors that combine dynamic scheduling and static scheduling
    - Itanium Processor Family (based on IA-64 ISA)
      - Explicit Parallel Instruction Computing (EPIC)

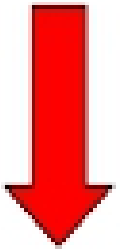
How to Keep a Pipeline Full?

- Exploit Parallelism “in Time”
  - find sequences of independent instructions that can be overlapped in the pipeline
- Scheduling problem for a compiler
  - given “source instruction” i and “dependent instruction” j
  - avoid pipeline stalls by…
  - …separating j from i by a distance in clock cycles that is equal to the latency of the pipeline execution of i



Example: a Simple Parallel Loop

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



compiled into MIPS assembly language

```
loop:   L.D      F0, 0(R1)      ; R1 = highest address of array element
        ADD.D    F4, F0, F2     ; add scalar s (stored in F2)
        S.D      F4, 0(R1)      ; store result
        DADDUI   R1,R1,#-8      ; decrement pointer by 8 bytes (DW)
        BNE      R1,R2, loop    ; 8(R2) points to x[1], last to process
```

Assumptions for Following Examples: Latencies and Resources of MIPS Pipeline

instruction Producing result	instruction Consuming result	Latency in Clock cycles
FP ALU op	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	1

- Additional assumption: no structural hazards
  - functional units are fully pipelined (or replicated as many times as the pipeline depth) so that an operation of any type can be issued on every clock cycle

Example: Scheduling the Simple Parallel Loop on the MIPS Pipeline

```
loop:   L.D      F0, 0(R1)     ; R1 = highest address of array element
        ADD.D    F4, F0, F2    ; add scalar s (which is stored in F2)
        S.D      F4, 0(R1)     ; store result
        DADDUI   R1,R1,#-8     ; decrement pointer by 8 bytes (DW)
        BNE      R1,R2, loop; 8(R2) points to x[1], last to process
```

- unscheduled execution

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

- executionTime = 9 cycles

- scheduled execution

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

- executionTime = 7 cycles

- **NOTE: execution time is expressed in 'clock cycles per loop iteration '**

Scheduled Execution and Critical Path

- The clock cycle count for the loop is determined by the critical path, longest chain of dependent instructions
  - in the example, 6 clock cycles to execute sequentially the chain

```
L.D      F0, 0(R1)
stall
ADD.D    F4, F0, F2
stall
stall
S.D      F4, 8(R1)
```

- A good compiler (capable of some symbolic optimization) optimizes the "filling" of those stalls, for instance by altering and interchanging **S.D** with **DADDUI**

- scheduled execution

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

- executionTime = 7 cycles

Loop Overhead and Loop Unrolling

- At each iteration of the loop, the only actual work is done on the array

```
L.D     F0,  0(R1)
ADD.D   F4,  F0,  F2
S.D     F4,  8(R1)
```

- Beside the stall cycle, **DADDUI** and **BNE** represent loop overhead cycles
- To reduce loop overhead, before scheduling the loop execution apply loop unrolling
  1. replicate the loop body multiple times
  2. merge unnecessary instructions
  3. eliminate the name dependencies
  4. adjust the control code

• scheduled execution

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

• executionTime = 7 cycles

Reducing Loop Overhead by Loop Unrolling before Scheduling – Step 1 (replication)

• original loop

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

• unrolling the original loop 4 times (dropping 3 **BNE**)

```
loop:   L.D      F0,   0(R1)
        ADD.D    F4,   F0,  F2
        S.D      F4,   0(R1)
        DADDUI   R1,   R1,  #-8

        L.D      F0,   0(R1)
        ADD.D    F4,   F0,  F2

        S.D      F4,   0(R1)
        DADDUI   R1,   R1,  #-8
        L.D      F0,   0(R1)
        ADD.D    F4,   F0,  F2
        S.D      F4,   0(R1)
        DADDUI   R1,   R1,  #-8

        L.D      F0,   0(R1)
        ADD.D    F4,   F0,  F2
        S.D      F4,   0(R1)
        DADDUI   R1,   R1,  #-8
        BNE      R1,   R2,  loop
```

Execution Time after Step 1

- unrolling the original loop 4 times (dropping 3 **BNE**)

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

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

- executionTime = 33 cycles

Reducing Loop Overhead by Loop Unrolling before Scheduling – Step 2 (merging)

• unrolling the original loop 4 times (dropping 3 **BNE**)

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

• merging **DADDUI** and adjusting the offsets

```
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)
       DADDUI  R1,  R1,  #-32
       BNE     R1,  R2,  loop
```

Execution Time after Step 2

- merging **DADDUI** and adjusting the offsets

```
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)
        DADDUI   R1,  R1,  #-32
        BNE      R1,  R2,  loop
```

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

- executionTime = 27 cycles

Reducing Loop Overhead by Loop Unrolling before Scheduling – Step 3 (renaming)

• merging **DADDUI** and adjusting the offsets

```
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)
       DADDUI   R1,  R1,  #-32
       BNE      R1,  R2,  loop
```

• eliminating name dependences via register renaming

```
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)
       DADDUI   R1,  R1,  #-32
       BNE      R1,  R2,  loop
```

Execution Time after Step 3

- eliminating name dependences via register renaming

```
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)
        DADDUI   R1,   R1,   #-32
        BNE      R1,   R2,   loop
```

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

- executionTime = 27 cycles

Reducing Loop Overhead by Loop Unrolling before Scheduling – Step 4 (scheduling)

- eliminating name dependences via register renaming

```
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)
        DADDUI   R1,   R1,   #-32
        BNE      R1,   R2,   loop
```

- scheduling

```
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)
        DADDUI   R1,   R1,   #-32
        S.D      F12,  16(R1)
        S.D      F16,  8(R1)
        BNE      R1,   R2,   loop
```

Scheduling after Step 4

- scheduling

```
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)
        DADDUI   R1,   R1,   #-32
        S.D      F12,  16(R1)
        BNE      R1,   R2,   loop
        S.D      F16,  8(R1)
```

```
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)
        DADDUI   R1,   R1,   #-32
        S.D      F12,  16(R1)
        S.D      F16,  8(R1)
        BNE      R1,   R2,   loop
```

- executionTime = 14 cycles

Final Performance Comparison

• original scheduled execution

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

• executionTime = 7cycles
• execution Time Per Element = 7 cycles
• e.g., an array of 1000 elements is processed in 7,000 cycles

• scheduled execution after loop unrolling

```
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)
        DADDUI   R1,   R1,   #-32
        S.D      F12,  16(R1)
        S.D      F16,  8(R1)
        BNE      R1,   R2,   loop
```

• executionTime = 14 cycles
• execution Time Per Element = 3.5 cycles
• e.g., an array of 1000 elements is processed in 3,500 cycles

Loop-unrolling speedup = 2

Loop Unrolling & Scheduling: Summary

- How is the final instruction sequence obtained?
- The compiler needs to determine that it is both possible and convenient to…
  - unroll the loop because the iterations are independent (except for the loop maintenance code)
  - eliminate extra loop maintenance code as long as the offsets are properly adjusted
  - use different registers to increase parallelism by removing name dependencies
  - remove stalling cycles by interleaving the instructions (and adjusting the offset or index increment consequently)
    - e.g. the compiler analyzes the memory addresses to determine that loads and stores from different iterations are independent

Between Statically-Scheduled and Dynamically-Scheduled Superscalars

- Superscalar processors decide on the fly how many instructions to issue in a clock cycle
- Key is to detect any dependency
  - between candidate instructions for an issue packet
  - between any 'issue candidate' and any 'instruction already in the pipeline'
- Statically-Scheduled Superscalar Processors
  - rely on compiler assistance
- Dynamically-Scheduled Superscalar Processors
  - rely on specialized hardware
- Very Long Instruction Word (VLIW) Processors
  - let the compiler format the potential issue packet (possibly indicating that a dependency exists)

Multiple-Issue Processors

- Superscalars
  - issue a varying number of instructions per clock cycle
  - either statically scheduled by the compiler
    - using in-order execution
  - or dynamically scheduled by the hardware
    - using out-of-order execution (techniques like Scoreboard, Tomasulo's Algorithm…)
- VLIW (Very Long Instruction Words) Processors
  - issue a fixed number of instructions formatted either as
    - one large instruction, or
    - a fixed instruction packet with parallelism explicitly indicated by the instruction (EPIC for IA-64 in Itanium processors)
  - inherently statically scheduled by the compiler

Exemple: Two-Issue Statically-Scheduled MIPS Superscalar

- Assume a two-issue statically-scheduled MIPS Superscalar implementation that can issue up to two instructions per clock cycles, i.e.
  - 1 integer/memory/control instruction
  - 1 FP instruction

Inst.Type								
non-FP	IF	ID	EX	MEM	WB			
FP	IF	ID	EX	MEM	WB			
non-FP		IF	ID	EX	MEM	WB		
FP		IF	ID	EX	MEM	WB		
non-FP			IF	ID	EX	MEM	WB	
FP			IF	ID	EX	MEM	WB	
non-FP				IF	ID	EX	MEM	WB
FP				IF	ID	EX	MEM	WB

Executing the Simple Loop on a Two-Issue Statically-Scheduled MIPS Superscalar

Integer Instruction	FP Instruction	Clock Cycle
loop: L.D F0, 0(R1)		1
L.D F6, -8(R1)		2
L.D F10, -16(R1)	ADD.D F4, F0, F2	3
L.D F14, -24(R1)	ADD.D F8, F6, F2	4
L.D F18, -32(R1)	ADD.D F12, F10, F2	5
S.D F4, -0(R1)	ADD.D F16, F14, F2	6
S.D F8, -8(R1)	ADD.D F20, F18, F2	7
S.D F12, -16(R1)		8
DADDUI R1,R1,#-40		9
S.D F16, 16(R1)		10
BNE R1, R2, loop		11
S.D F20, 8(R1)		12

- unrolled loop 5 times
- execTime = 12 cycles
- execTimePerElement = 2.4 cycles
- superscalar speedup = 1.5 (with respect to loop unrolling)
- combined speedup = 2.5 (= 1.7 * 1.5)
- efficiency = =17/24 = 71%

VLIW: Very Long Instruction Word

- A “very long” (64-bit, 128-bit,…) instruction encodes several operations that can be executed in parallel on multiple independent functional units
  - packets in EPIC, molecules in Transmeta
- Same compiler/architecture concepts for either
  - multiple operations organized into a single instruction
  - set of instructions grouped by the compiler into a single packet while excluding dependencies
- The burden of detecting dependencies falls on the compiler (simpler HW than in a Superscalar)
- VLIW is effective with wide-issue processors
  - more than 2-way issue
  - 16/24 bits per field (112/168 bits of instruction length)

VLIW vs. Superscalars

- VLIW executes operations in parallel based on a fixed schedule determined when programs are compiled
  - no scheduling hardware like in superscalars
  - increase computational power with less hardware complexity (but greater compiler complexity) than is associated with most superscalar CPUs
- One VLIW instruction encodes multiple operations, at best one operation for each execution unit in the data path
  - in superscalar designs, instead, the number of execution units is invisible to the instruction set
    - each instruction encodes only one operation

Example: Executing the Simple Loop on a MIPS VLIW Processor

- Assuming a VLIW that can issue up to 2 memory references, 2 FP operations and 1 integer/branch operation per clock cycle

Memory Ref. 1	Memory Ref. 2	FP 1	FP Instruction	Other
L.D F0, 0(R1)	L.D F6, -8(R1)			
L.D F10, -16(R1)	L.D F14, -24(R1)			
L.D F18, -32(R1)	L.D F22, -40(R1)	ADD.D F4, F0, F2	ADD.D F8, F6, F2	
L.D F26, -48(R1)		ADD.D F12, F10, F2	ADD.D F16, F14, F2	
		ADD.D F20, F18, F2	ADD.D F24, F22, F2	
S.D F4, -0(R1)	S.D F8, -8(R1)	ADD.D F28, F26, F2		
S.D F12, -16(R1)	S.D F16, -24(R1)			DADDUI R1,R1,#-56
S.D F20, 24(R1)	S.D F24, 16(R1)			
S.D F28, 8(R1)				BNE R1,R2, loop

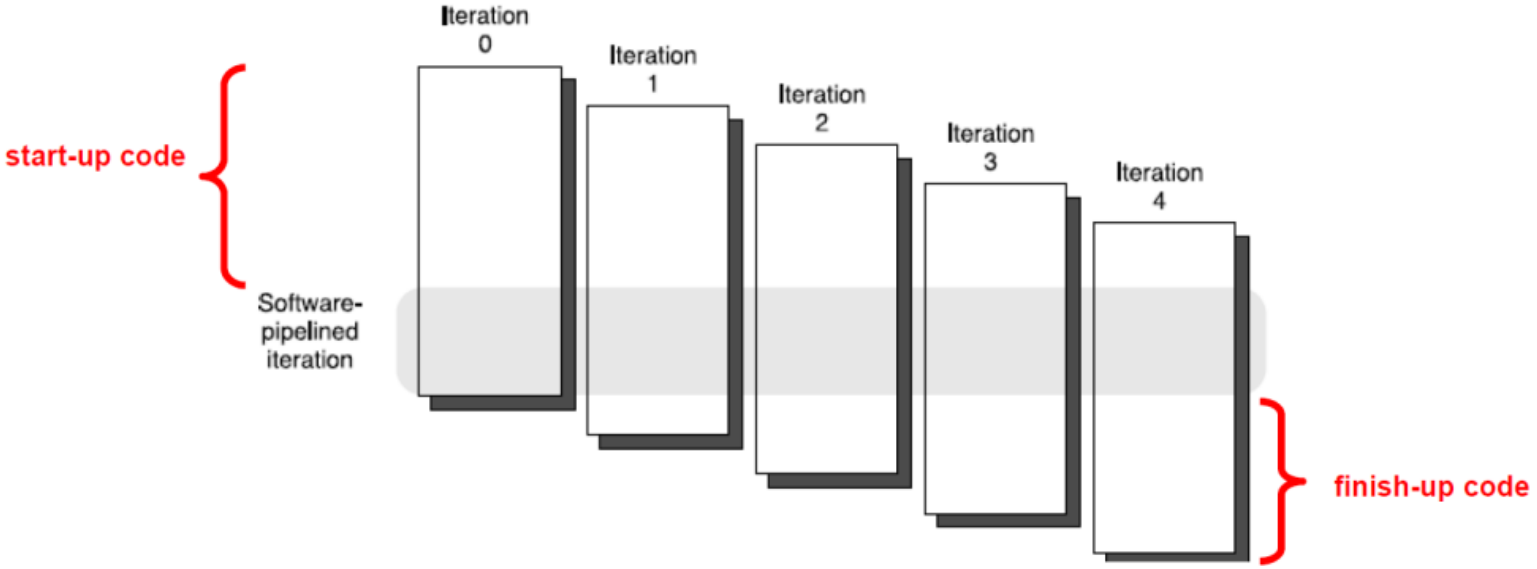
- unrolled loop 7 times (this eliminates stalls, i.e. empty issue cycles)
- execTime=9cycles
- execTimePerElement=9/7=1.29cycles
- VLIW speedup=2.71(with respect to loop unrolling)
- combined speedup(with loop unrolling)=4.6(=1.7*2.71)
- efficiency(% of busy slots)=23/45=51%

VLIWs: Techniques to Increasing Efficiency

- To keep functional units busy there must be enough parallelism in a code sequence
  - loop unrolling & scheduling
    - expand a loop by grouping instructions from multiple iterations and interleaving them
  - software pipelining & scheduling
    - keep the same number of instructions per loop but take instructions from different iterations and interleave them
  - local scheduling
    - if unrolling generates straight-line code
  - global scheduling
    - if necessary to schedule across branches
      - structurally more complex and harder to optimize

Software Pipelining

- Observation: if iterations from loops are independent, then can get ILP by taking instructions from different iterations
- Software pipelining: interleave instructions from different loop iterations
  - reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (Tomasulo in SW)



Software Pipelining Example

```
LOOP: 1 L.D     F0,0(R1)
      2 ADD.D   F4,F0,F2
      3 S.D     F4,0(R1)
      4 DADDUI  R1,R1,#8
      5 BNE     R1,R2,LOOP
```

(5/5)

2 RAWs!

```
 1 L.D    F0,0(R1)
 2 ADD.D  F4,F0,F2   4 L.D    F0,-8(R1)
 3 S.D    F4,0(R1)   5 ADD.D  F4,F0,F2   7 L.D    F0,-16(R1)
                     6 S.D    F4,-8(R1)  8 ADD.D  F4,F0,F2
                                         9 S.D    F4,-16(R1)
```

Before: Unrolled 3 times (11/11)

```
1   L.D      F0,0(R1)
2   ADD.D    F4,F0,F2
3   S.D      F4,0(R1)
4   L.D      F0,-8(R1)
5   ADD.D    F4,F0,F2
6   S.D      F4,-8(R1)
7   L.D      F0,-16(R1)
8   ADD.D    F4,F0,F2
9   S.D      F4,-16(R1)
10  DADDUI   R1,R1,#24
11  BNE      R1,R2,LOOP
```

After: Software Pipelined (5/11)

```
1  L.D     F0,0(R1)
2  ADD.D   F4,F0,F2
4  L.D     F0,-8(R1)
3  S.D     F4,0(R1);    Stores M[i]
5  ADD.D   F4,F0,F2;    Adds to M[i-1]
7  L.D     F0,-16(R1); loads M[i-2]
10 DADDUI  R1,R1,#8
11 BNE     R1,R2,LOOP
6  S.D     F4,-8(R1)
8  ADD.D   F4,F0,F2
9  S.D     F4,-16(R1)
```

No RAW! (2 WARs)

Software Pipelining: Summary

- Major advantage over straight loop unrolling
  - SW pipelining consumes less code space
- Can be applied together with loop unrolling
  - they reduce different types of overhead
- Compilation using SW pipelining is quite difficult
  - many loops require significant transformations
  - trade-offs in terms of overhead versus efficiency of the SW-pipelined loop are complex to analyze
  - register management can be tricky
- Some modern processors add specialized HW support for SW pipelining
  - Intel IA-64 ISA