Στατικές Αρχιτεκτονικές Παραλληλισμού Επιπέδου Εντολής – Τεχνικές Λογισμικού

Νεκτάριος Κοζύρης & Διονύσης Πνευματικάτος
{nkoziris,pnevmati}@cslab.ece.ntua.gr

8ο εξάμηνο ΣΗΜΜΥ – Ακαδημαϊκό Έτος: 2019-20

http://www.cslab.ece.ntua.gr/courses/advcomparch/

Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to maximize IPC (or minimize CPI)
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls
- Parallelism within *basic block* is limited
  - A code sequence:
    - no branch in except to the entry & no branch out except at the exit
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches

Data Dependence

- Loop-Level Parallelism (plenty in many cases)
  - Unroll loop statically or dynamically
  - Use SIMD (vector processors and GPUs)

- Challenges:
  - Data dependency
    - Instruction *j* is data dependent on instruction *i* if
      - Instruction *i* produces a result that may be used by instruction *j*
      - Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i*

- Dependent instructions cannot be executed simultaneously

Getting CPI below 1 (superscalar)

- CPI ≥ 1 if issue only 1 instruction every clock cycle
- Multiple-issue processors come in 3 flavors:
  - statically-scheduled superscalar processors,
  - dynamically-scheduled superscalar processors, and
  - VLIW (very long instruction word) processors
  - Also, Vector, SIMD…
- 2 types of superscalar processors issue varying numbers of instructions per clock
  - use in-order execution if they are statically scheduled, or
  - out-of-order execution if they are dynamically scheduled
- VLIW processors, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism within instructions explicitly indicated by the instruction (Intel/HP Itanium)

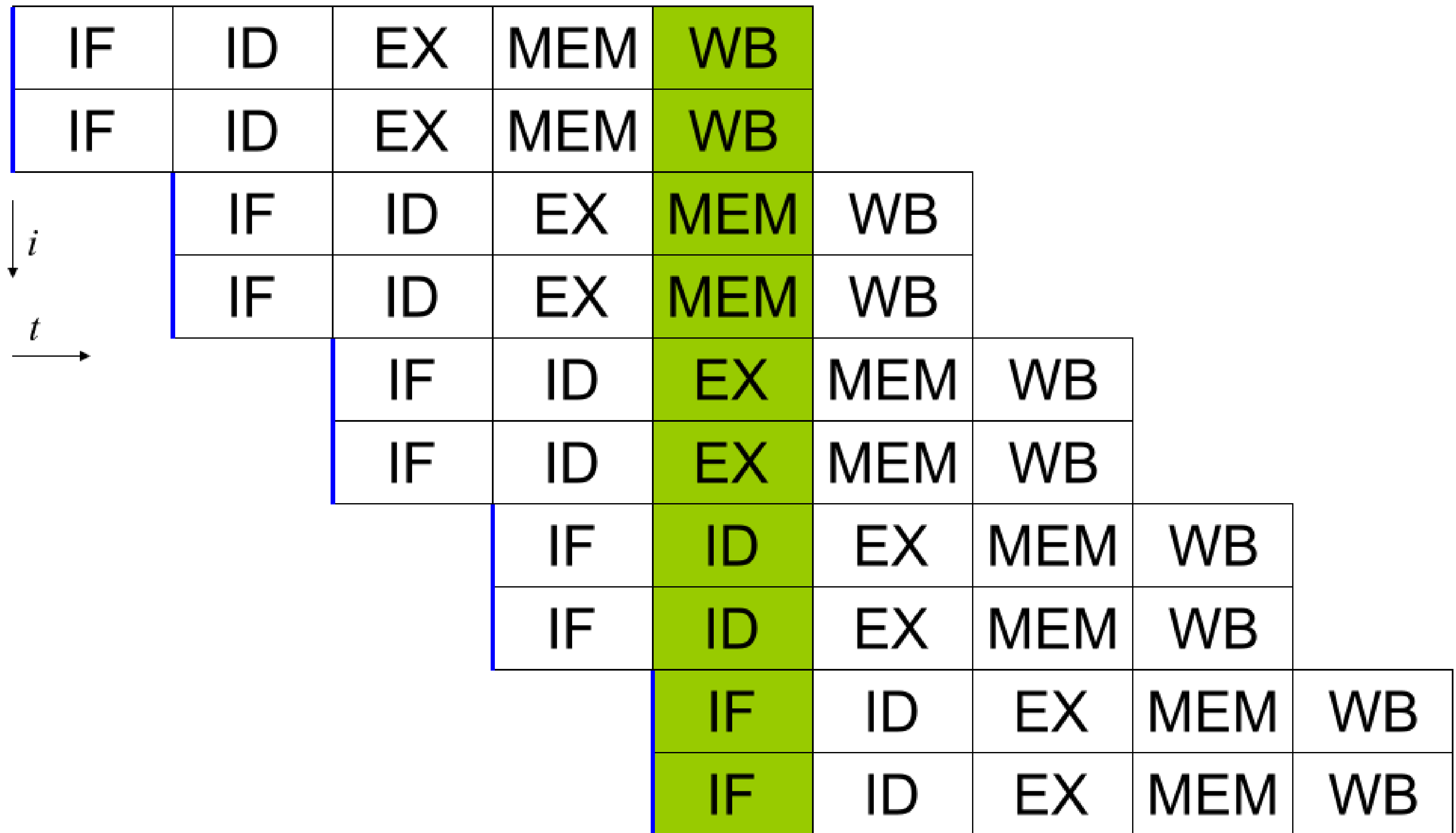
Static Multiple Issue

- Issue >1 instructions per cycle (multiple parallel pipelines)
- Examine all the dependencies among the instructions in the bundle
- If no dependencies exist in bundle, execute
- If there are, issue only independent

Easier/cheaper

- Limit the classes of instructions that can be parallel
  - i.e. one FP, one scalar, etc

Static Multiple Issue



Example

```
Loop:  LD R2,0(R1)            ;R2=array element
       DADDIU R2,R2,#1        ;increment R2
       SD R2,0(R1)            ;store result
       DADDIU R1,R1,#8        ;increment pointer
       BNE R2,R3,LOOP         ;branch if not last element
```

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Superscalar MIPS: 2 instructions, 1 FP & 1 anything
  – Fetch 64-bits/clock cycle; Int on left, FP on right
  – Can only issue 2nd instruction if 1st instruction issues
  – More ports for FP registers to do FP load & FP op in a pair

Type	*Pipe Stages*						
Int. instruction	IF	ID	EX	MEM	WB		
FP instruction	IF	ID	EX	MEM	WB		
Int. instruction		IF	ID	EX	MEM	WB	
FP instruction		IF	ID	EX	MEM	WB	
Int. instruction			IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB

- 1 cycle load delay expands to 3 instructions in SS
  - instruction in right half can't use it, nor instructions in next slot

Multiple Issue Details

- issue packet: group of instructions from fetch unit that could potentially issue in 1 clock
  - If instruction causes structural or a data hazard either due to earlier instruction in execution or to earlier instruction in issue packet, then instruction cannot issue
  - 0 to N instruction issues per clock cycle, for N-issue
- Performing issue checks in 1 cycle could limit clock cycle time: $O(n^2-n)$ comparisons
  - => issue stage usually split and pipelined
  - 1st stage decides how many instructions from within this packet can issue, 2nd stage examines hazards among selected instructions and those already been issued
  - => higher branch penalties => prediction accuracy important

Multiple Issue Challenges

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations AND No hazards
- If more instructions issue at same time, greater difficulty of decode and issue:
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue; (N-issue ~$O(N^2-N)$ comparisons)
  - Register file: need 2xN reads and 1xN writes/cycle
  - Rename logic: must be able to rename same register multiple times in one cycle (multiple registers too)! Consider 4-way issue:

    ```
    add r1, r2, r3              add p11, p4, p7
    sub r4, r1, r2      ⇒       sub p22, p11, p4
    lw  r1, 4(r4)               lw  p23, 4(p22)
    add r5, r1, r2              add p12, p23, p4
    ```

    Imagine doing this transformation in a single cycle!
  - Result buses: Need to complete multiple instructions/cycle
    - So, need multiple buses with associated matching logic at every reservation station.
    - Or, need multiple forwarding paths

Compiler Techniques for Exposing ILP

- Pipeline scheduling
  - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction
- Example:

```
for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;
```

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

FP Loop: Where are the Hazards?

```
for (i=999; i>=0; i=i-1)
   x[i] = x[i] + s;
```

To simplify assume:

- 8 is lowest address
- R1 = base address of X
- F2 = s

```
Loop:L.D     F0,0(R1) ;F0=vector element
    ADD.D    F4,F0,F2 ;add scalar from F2
    S.D      0(R1),F4 ;store result
    DADDUI   R1,R1,-8 ;decrement pointer 8B(DW)
    BNEZ     R1,Loop  ;branch R1!=zero
```

FP Loop Showing Stalls

```
1 Loop: L.D F0,0(R1)        ;F0=vector element
2      stall
3      ADD.D  F4,F0,F2      ;add scalar in F2
4      stall
5      stall
6      S.D    0(R1),F4      ;store result
7      DADDUI R1,R1,-8 ;decrement pointer 8B (DW)
8      stall     ;assumes can't forward to branch
9      BNEZ    R1,Loop      ;branch R1!=zero
```

Cost: 9 clock cycles

Can we rewrite the code to minimize stalls?

Revised FP Loop Minimizing Stalls

Swap DADDUI and S.D by changing address of S.D

```
1 Loop: L.D   F0,0(R1)
2       DADDUI R1,R1,-8
3       ADD.D  F4,F0,F2
4       stall
5       stall
6       S.D    8(R1),F4 ;fix offset after DADDUI
7       BNEZ R1,Loop
```

7 clock cycles: 3 for execution (L.D, ADD.D,S.D) and 4 for loop overhead. How to make faster?

Unroll Loop Four Times (Simple way)

```
1  Loop:L.D    F0,0(R1)
3       ADD.D  F4,F0,F2
6       S.D    0(R1),F4     ;drop DSUBUI & BNEZ
7       L.D    F6,-8(R1)
9       ADD.D  F8,F6,F2
12      S.D    -8(R1),F8    ;drop DSUBUI & BNEZ
13      L.D    F10,-16(R1)
15      ADD.D  F12,F10,F2
18      S.D    -16(R1),F12  ;drop DSUBUI & BNEZ
19      L.D    F14,-24(R1)
21      ADD.D  F16,F14,F2
24      S.D    -24(R1),F16
25      DADDUI          R1,R1,#-32   ;alter to 4*8
26      BNEZ   R1,LOOP
```

1 cycle stall

2 cycles stall

Rewrite loop to minimize stalls?

Assumes R1 is multiple of 4: ***14 instructions vs 28!***

27 clock cycles, or 6.75 per iteration

Unrolled Loop That Minimizes Stalls

```
1  Loop:L.D F0,0(R1)
2       L.D     F6,-8(R1)
3       L.D     F10,-16(R1)
4       L.D     F14,-24(R1)
5       ADD.D   F4,F0,F2
6       ADD.D   F8,F6,F2
7       ADD.D   F12,F10,F2
8       ADD.D   F16,F14,F2
9       S.D     0(R1),F4
10      S.D     -8(R1),F8
11      S.D     -16(R1),F12
12      DSUBUI  R1,R1,#32
13      S.D     8(R1),F16 ; 8-32 = -24
14      BNEZ    R1,LOOP
```

No stalls?

14 instructions, 14 clock cycles, or 3.5 per iteration

Unrolled Loop Details

- Do not usually know upper bound of loop
- Suppose it is n, and we would like to unroll the loop to make k copies of the body
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
  - 1st executes (n mod k) times and has a body that is the original loop
  - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
- For large values of n, most of the execution time will be spent in the unrolled loop
- Called "Strip mining"
- *Versioning* the code: different sequences for different cases/conditions/…

5 Loop Unrolling Decisions

Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:

- Determine loop unrolling useful by finding that loop iterations were independent (except for maintenance code)
- Use different registers to avoid unnecessary constraints forced by using same registers for different computations
- Eliminate the extra test and branch instructions and adjust the loop termination and iteration code
- Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent
  - Transformation requires analyzing memory addresses and finding that they do not refer to the same address
- Schedule the code, preserving any dependences needed to yield the same result as the original code

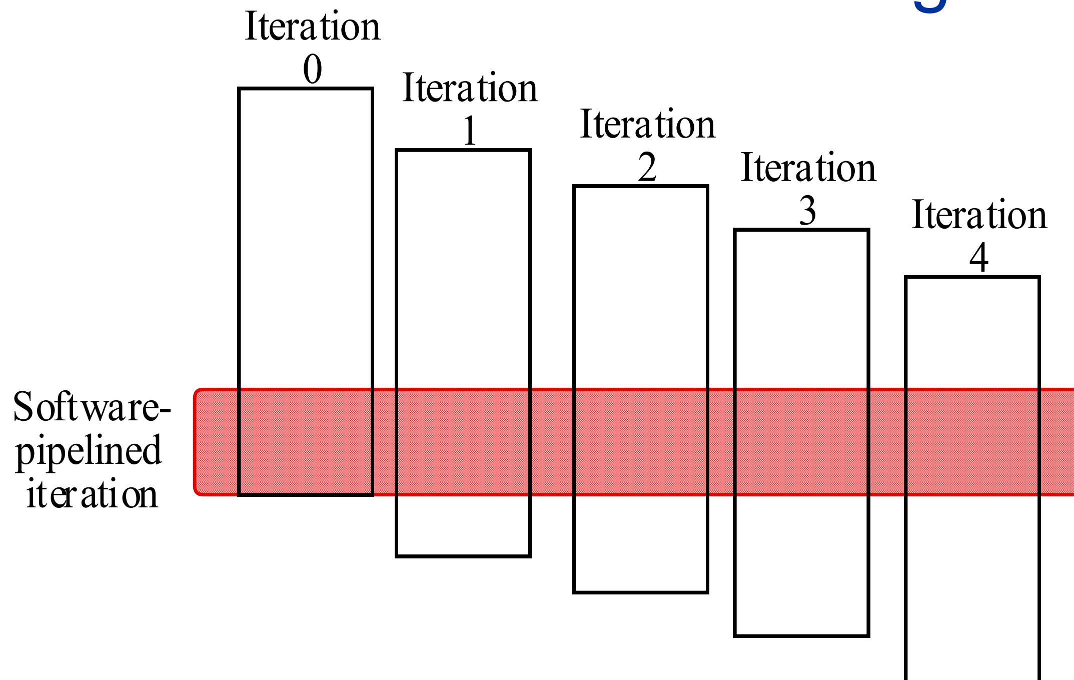
3 Limits to Loop Unrolling

1) Decrease in amount of overhead amortized with each extra unrolling
   - Remember Amdahl's Law
2) Growth in code size
   - For larger loops, it increases the instruction cache miss rate
   - Code size important for embedded devices
3) Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling
   - If not possible to allocate all live values to registers, may lose some or all of its advantage

- Loop unrolling reduces impact of branches on pipeline; another way is branch prediction

Another possibility: Software Pipelining

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



S/W Pipelining: prolog/epilogue

```
for i = 1 to bignumber
    A(i)
    B(i)
    C(i)
End
```

Main loop body very small!

Prologue: A(1), A(2), B(1)

```
for i = 1 to (bignumber – 2)
  A(i+2)
  B(i+1)
  C(i)
End
```

Epilogue: B(10) C(9), C(10)

Assume: A, B, C independent

SW Pipelining

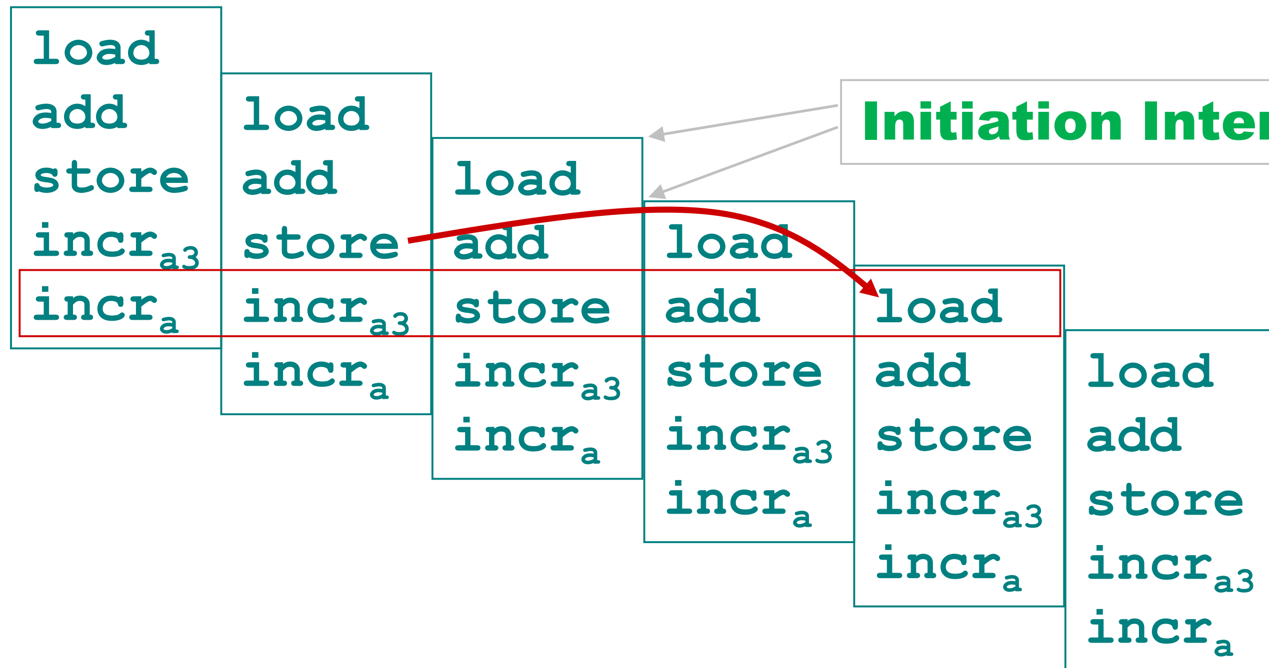
Source code:

```
for(i=2; i<n;i++)
   a[i] = a[i-3] + c;
```

dependence spans three iterations
"distance = 3"

Assembly:

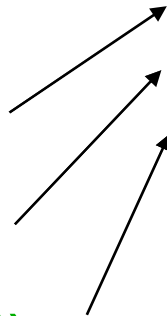
```
load_a
add
store
incr_a3
incr_a
```



Software Pipelining Example

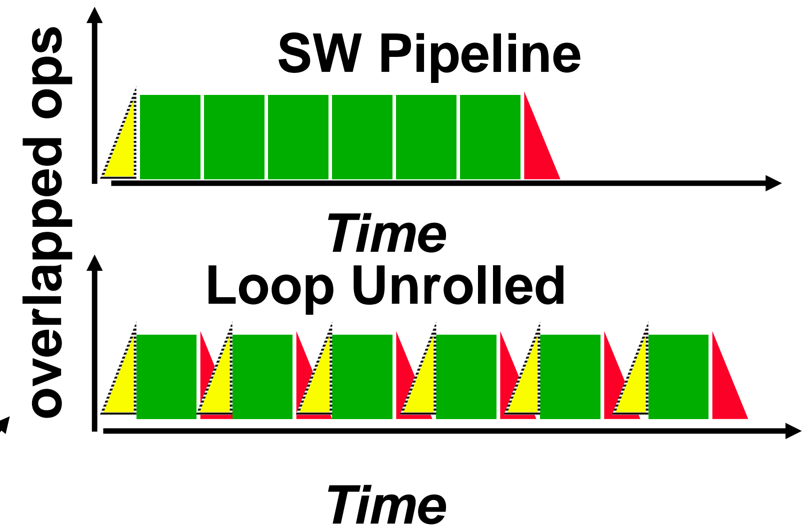
Before: Unrolled 3 times

```
1   L.D     F0,0(R1)
2   ADD.D   F4,F0,F2
3   S.D     0(R1),F4
4   L.D     F6,-8(R1)
5   ADD.D   F8,F6,F2
6   S.D     -8(R1),F8
7   L.D     F10,-16(R1)
8   ADD.D   F12,F10,F2
9   S.D     -16(R1),F12
10  DSUBUI  R1,R1,#24
11  BNEZ    R1,LOOP
```



After: Software Pipelined

```
1   S.D     0(R1),F4 ;     Stores M[i]
2   ADD.D   F4,F0,F2 ;     Adds to M[i-1]
3   L.D     F0,-16(R1);    Loads M[i-2]
4   DSUBUI  R1,R1,#8
5   BNEZ    R1,LOOP
```



Symbolic Loop Unrolling

- Maximize result-use distance
- Less code space than unrolling
- Fill & drain pipe only once per loop
  vs. once per each unrolled iteration in loop unrolling

5 cycles per iteration

When Safe to Unroll Loop?

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

```
for (i=0; i<100; i=i+1) {
        A[i+1] = A[i] + C[i];      /* S1 */
        B[i+1] = B[i] + A[i+1];    /* S2 */
}
```

- S2 uses the value A[i+1] computed by S1 in the same iteration
- 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
- For the previous example each iteration was distinct
- Implies that iterations can't be executed in parallel, right????

Loop-carried dependence => no parallelism?

- Consider:

```
for (i=0; i< 8; i=i+1) {
        A = A + C[i];    /* S1 */
}
```

Could compute:

```
"Cycle 1":    temp0 = C[0] + C[1];
              temp1 = C[2] + C[3];
              temp2 = C[4] + C[5];
              temp3 = C[6] + C[7];
"Cycle 2":    temp4 = temp0 + temp1;
              temp5 = temp2 + temp3;
"Cycle 3":  A = temp4 + temp5;
```

- Relies on associative nature of “+”.
- See “Parallelizing Complex Scans and Reductions” by Allan Fisher and Anwar Ghuloum

VLIW Processors

- Package multiple operations into one instruction
- Example VLIW processor:
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references

Memory Reference #1	Memory Reference #2	FP Operation #1	FP Operation #2	Integer Operation/Branch

- Must be enough parallelism in code to fill the available slots

VLIW Processors

- Disadvantages:
  - Statically finding parallelism
  - Code size (store even empty slots)
  - No hazard detection hardware
  - Binary code compatibility?
    - What happens to the code for a new, bigger, better model?

VLIW: Very Large Instruction Word

- Each "instruction" explicitly codes multiple operations
  - In IA-64, grouping called a "packet"
  - In Transmeta, grouping called a "molecule" (with "atoms" as ops)
- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Compiler must schedule across several branches

Recall: Unrolled Loop Minimizes Stalls for Scalar

```
1  Loop:   L.D      F0,0(R1)
2          L.D      F6,-8(R1)
3          L.D      F10,-16(R1)
4          L.D      F14,-24(R1)
5          ADD.D    F4,F0,F2
6          ADD.D    F8,F6,F2
7          ADD.D    F12,F10,F2
8          ADD.D    F16,F14,F2
9          S.D      0(R1),F4
10         S.D      -8(R1),F8
11         S.D      -16(R1),F12
12         DSUBUI   R1,R1,#32
13         BNEZ     R1,LOOP
14         S.D      8(R1),F16    ; 8-32 = -24
```

L.D to ADD.D: 1 Cycle
ADD.D to S.D: 2 Cycles

14 clock cycles, or 3.5 per iteration

Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
L.D F0,0(R1)	L.D F6,-8(R1)				1
L.D F10,-16(R1)	L.D F14,-24(R1)				2
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2		3
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2		4
		ADD.D F20,F18,F2	ADD.D F24,F22,F2		5
S.D 0(R1),F4	S.D -8(R1),F8	ADD.D F28,F26,F2			6
S.D -16(R1),F12	S.D -24(R1),F16				7
S.D -32(R1),F20	S.D -40(R1),F24			DSUBUI R1,R1,#48	8
S.D -0(R1),F28				BNEZ R1,LOOP	9

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

Note: Need more registers in VLIW (15 vs. 6 in SS)

Problems with 1st Generation VLIW

- Increase in code size
  - generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding
- Operated in lock-step; no hazard detection HW
  - a stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  - Compiler might predict function units, but caches hard to predict
- Binary code compatibility
  - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code

Solutions to these Problems

- Smaller “packets” that express independence
  - Code Efficiency (stop early if no independent instructions)
  - Portability (can express too much parallelism that a future implementation can exploit)
- More registers, special registers, compiler techniques
  - Better code (utilization)
- Dynamic decisions
  - No lock-step, but more complex!

Multiple Issue

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium