COSC4201
Hardware Speculation and More ILP

Prof. Mokhtar Aboelaze

Parts of these slides are taken from Notes by
Prof. David Patterson (UCB)

Outline

- Data dependence and hazards
- Exposing parallelism (loop unrolling and scheduling)
- Reducing branch costs (prediction)
- Dynamic scheduling
- Speculation
- Multiple issue and static scheduling
- Advanced techniques
- Example

Introduction

- Loads or a stores can safely be done in any order, provided they access different addresses.
- If a load and a store access the same address, then
  - Either load is before store in program order, interchanging them results in WAR hazard.
  - The store is before the load in program order, interchanging them result in a RAW Hazard
  - Interchanging 2 stores, result in a WAW hazard.
- To proceed with a load, processor must check whether any uncompleted store that precedes the load in program order share the same data memory address as the load.
- Similarly, a store must check loads and stores.
- A not very efficient way, is to guarantee that address calculation are done in program order.

Speculation

- In dynamic scheduling, we wait before executing an instruction after a branch until the branch is resolved (integer operations may go ahead beyond branches).
- 3 components of HW-based speculation:

1. Dynamic branch prediction to choose which instructions to execute
2. Speculation to allow execution of instructions before control dependences are resolved

   + ability to undo effects of incorrectly speculated sequence

3. Dynamic scheduling to deal with scheduling of different combinations of basic blocks

Speculation

- Must separate execution from allowing instruction to finish or “commit”
- This additional step called instruction commit
- When an instruction is no longer speculative, allow it to update the register file or memory
- Requires additional set of buffers to hold results of instructions that have finished execution but have not committed
- This reorder buffer (ROB) is also used to pass results among instructions that may be speculated

---

Speculation

- In Tomasulo’s algorithm, once an instruction writes its result, any subsequently issued instructions will find result in the register file
- With speculation, the register file is not updated until the instruction commits
  - (we know definitively that the instruction should execute)
- Thus, the ROB supplies operands in interval between completion of instruction execution and instruction commit
  - ROB is a source of operands for instructions, just as reservation stations (RS) provide operands in Tomasulo’s algorithm
  - ROB extends architecture registers like RS
- ROB holds the results between the operation associated with the instruction completes, and commit

ROB

° **Each entry in the ROB contains four fields:**

1.Instruction type

- **a branch (has no destination result), a store (has a memory address destination), or a register operation (ALU operation or load, which has register destinations)**

2.Destination

- **Register number (for loads and ALU operations) or memory address (for stores)**
  where the instruction result should be written

3.Value

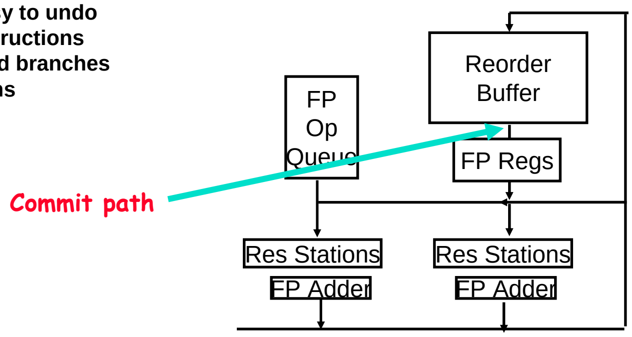
- **Value of instruction result until the instruction commits**

4.Ready

- **Indicates that instruction has completed execution, and the value is ready**

ROB

- **Holds instructions in FIFO order, exactly as issued**
- **When instructions complete, results placed into ROB**
  - **Supplies operands to other instruction between execution complete & commit ⇒ more registers like RS**
  - **Tag results with ROB buffer number instead of reservation station**
- **Instructions commit ⇒values at head of ROB placed in registers**
- **As a result, easy to undo speculated instructions on mispredicted branches or on exceptions**



Steps

1. **Issue**—get instruction from FP Op Queue

   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called "dispatch"), OR stall

2. **Execution**—operate on operands (EX)

   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called "issue")

3. **Write result**—finish execution (WB)

   Write on Common Data Bus to all awaiting FUs (ROB tag) & reorder buffer; mark reservation station available.

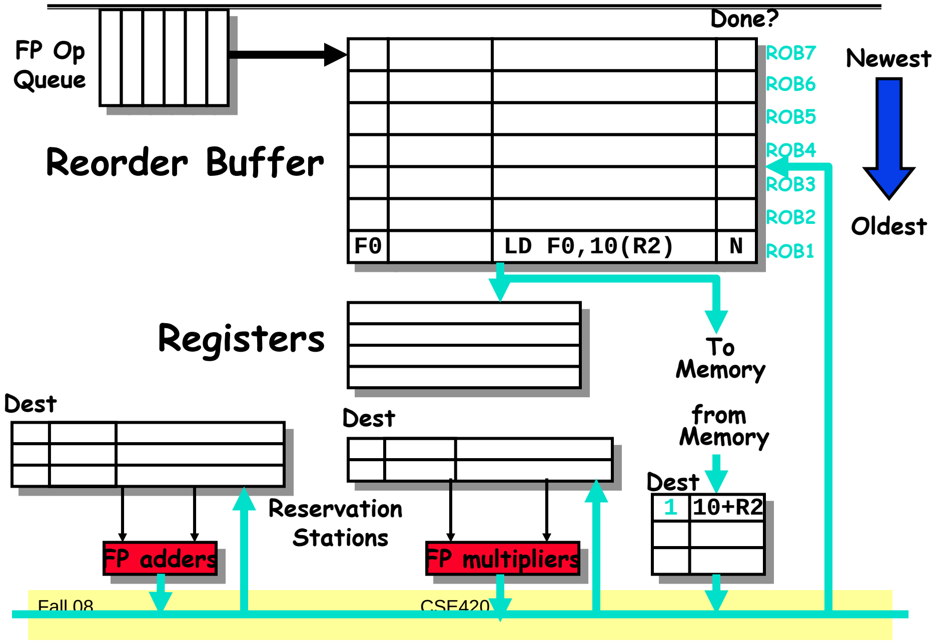
4. **Commit**—update register with reorder result

   When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called "graduation")

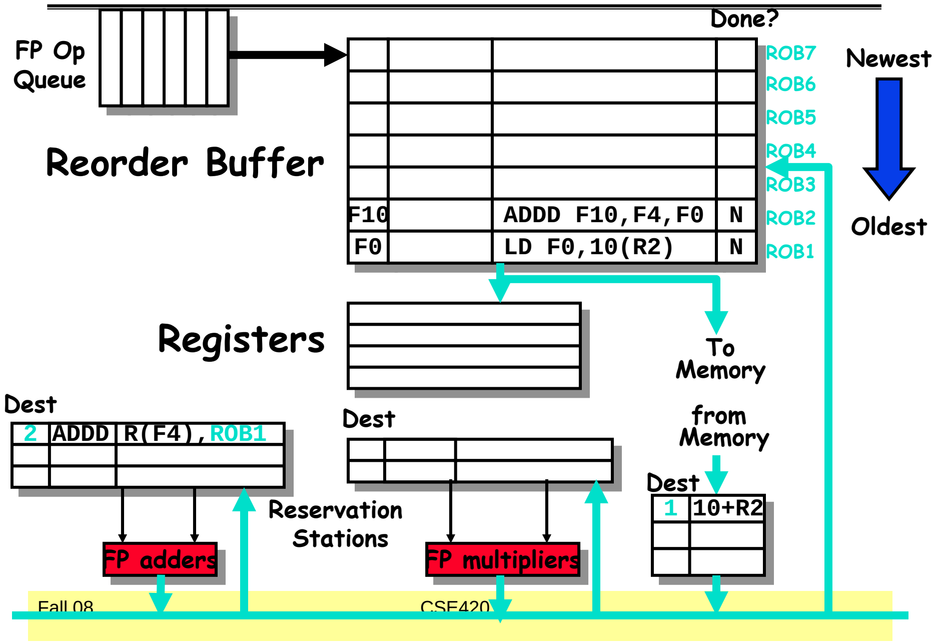
Example

```
Loop  LD          F0,10(R2)
      ADDD        F10,F4,F0
      DIVD        F2,F10,F6
      DADD        R1,R1,-8
      BNE         R1,R2,Loop
```

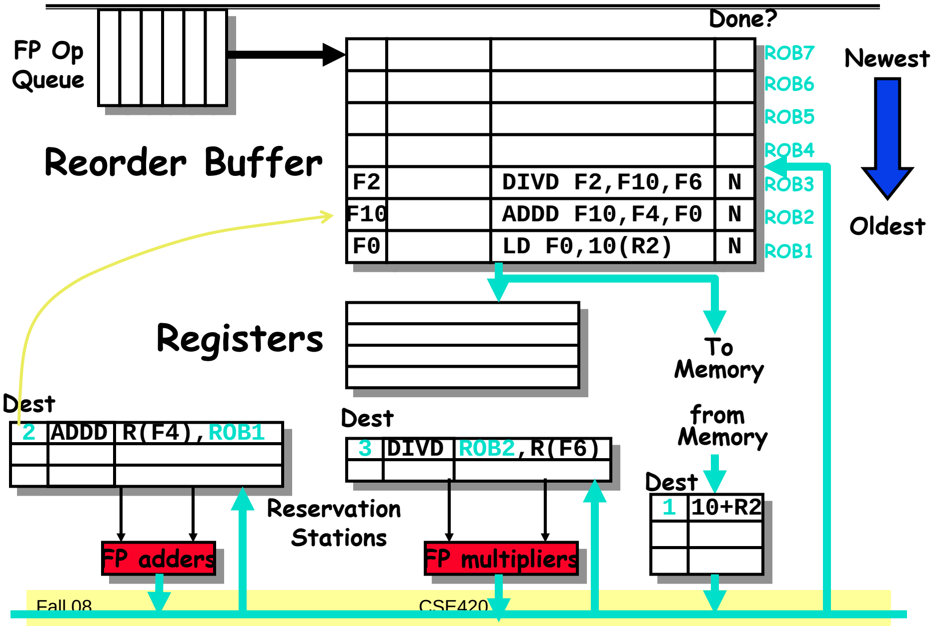
Tomasulo With Reorder buffer:



Tomasulo With Reorder buffer:



Tomasulo With Reorder buffer:



Outline

- Data dependence and hazards
- Exposing parallelism (loop unrolling and scheduling)
- Reducing branch costs (prediction)
- Dynamic scheduling
- Speculation
- Multiple issue and static scheduling
- Advanced techniques
- Example

VLIW

- **Each "instruction" has explicit coding for 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
  - **Need compiling technique that schedules across several branches**

VLIW -- Example

Source instruction	Instruction using result	Latency
FP ALU OP	FP ALU OP	3
FP ALU OP	Store double	2
Load double	FP ALU OP	1
Load Double	Store double	0

```
Loop: L.D          F0,0(R1)
      ADD.D        F4,F0,F2
      S.D          0(R1),F4
      DADDUI       R1,R1,#-8
      BNE  R       1,R2,Loop
```

For (I=1000;I>0;I++)

x[I]=x[I]+s;

VLIW -- Example

- Assume that w can schedule 2 memory operations, 2 FP operations, and one integer or branch

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2	3	
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2			6
SD -16(R1),F12	SD -24(R1),F16			DADD R1,R1,#-56	7
SD 24(R1),F20	SD 16(R1),F24				8
SD 8(R1),F28				BNEZ R1,LOOP	9

7 iterations in 9 cycles = 1.29 c/I

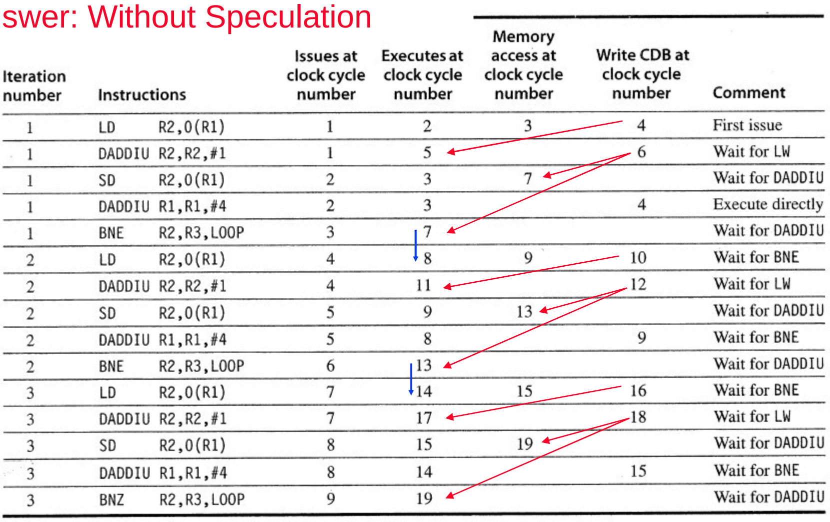
Advanced Dynamic Scheduling

- Dynamic Scheduling with multiple issue and speculation.
- Two different approaches
  - Issuing the instruction in half a cycle
  - Building the logic to issue 2 instructions simultaneously including detecting dependence
- Must be able to commit more than one instruction at the same time.

Advanced Dynamic Scheduling

```
Loop: LD    R2,0(R1)
      ADD   R2,R2,#1
      SD    R2,0(R1)
      ADD   R1,R1,#8
      BNE   R2,R3,Loop
```

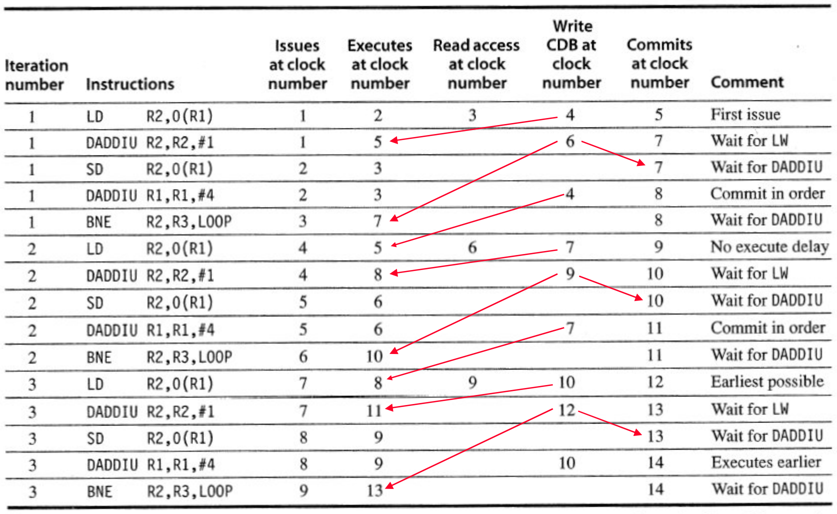
Answer: Without Speculation



Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#4	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#4	8	14		15	Wait for BNE
3	BNZ R2,R3,LOOP	9	19			Wait for DADDIU

Figure 3.33 The time of issue, execution, and writing result for a dual-issue version of our pipeline *without* speculation. Note that the L.D following the BNE cannot start execution earlier, because it must wait until the branch outcome is determined. This type of program, with data-dependent branches that cannot be resolved earlier, shows the strength of speculation. Separate functional units for address calculation, ALU operations, and branch condition evaluation allow multiple instructions to execute in the same cycle.

Answer: 2-way Superscalar Tomasulo With Speculation



Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LW
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LW
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#4	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#4	8	9		10	14	Executes earlier
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU

Figure 3.34 The time of issue, execution, and writing result for a dual-issue version of our pipeline *with* speculation. Note that the L.D following the BNE can start execution early because it is speculative.

Branches Still Single Issue

Loop Level Parallelism LLP

- **Loop-Level Parallelism (LLP) analysis focuses on whether data accesses in later iterations of a loop are data dependent on data values produced in earlier iterations and possibly making loop iterations independent.**
- **e.g. in for (i=1; i<=1000; i++)**

  x[i] = x[i] + s;

the computation in each iteration is independent of the previous iterations and the loop is thus parallel. The use of X[i] twice is within a single iteration.

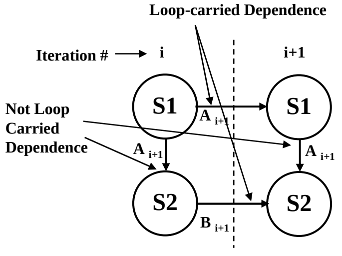
⇒ Thus loop iterations are parallel (or independent from each other).

Loop Level Parallelism LLP

- Loop-carried Dependence**: A data dependence between different loop iterations (data produced in earlier iteration used in a later one).**
- **LLP analysis is important in software optimizations such as loop unrolling since it usually requires loop iterations to be independent.**
- **LLP analysis is normally done at the source code level or close to it since assembly language and target machine code generation introduces loop-carried name dependence in the registers used for addressing and incrementing.**
- **Instruction level parallelism (ILP) analysis, on the other hand, is usually done when instructions are generated by the compiler**

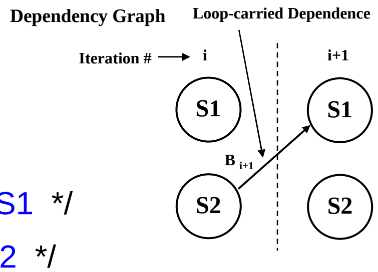
Loop Level Parallelism LLP

```
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 */
  }
```



Dependency Graph

- S2 uses the value A[i+1], computed by S1 in the same iteration. This data dependence is within the same iteration <u>(not a loop-carried dependence).</u>
  - ⇒ does not prevent loop iteration parallelism.
- S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] read in iteration i+1 <u>(loop-carried dependence, prevents parallelism).</u> The same applies for S2 for B[i] and B[i+1]
  - ⇒ These two dependencies are loop-carried spanning more than one iteration preventing loop parallelism.



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

- **S1 uses the value B[i] computed by S2 in the previous iteration (loop-carried dependence)**
- **This dependence is not circular:**
  - S1 depends on S2 but S2 does not depend on S1.

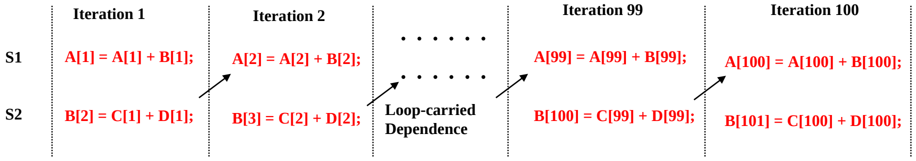
LLP Analysis Example 2

```
A[1] = A[1] + B[1];
  for (i=1; i<=99; i=i+1)  {
         B[i+1] = C[i] + D[i];
         A[i+1] = A[i+1] + B[i+1];
   }
   B[101] = C[100] + D[100];
```

LLP Analysis Example 2

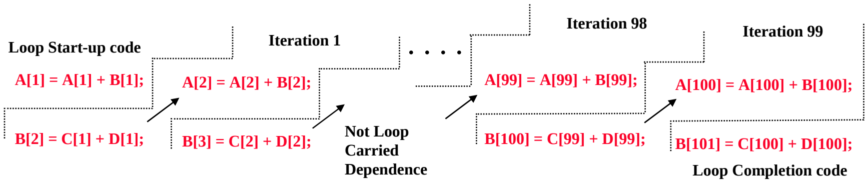
Original Loop:

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



Modified Parallel Loop:

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1)  {
      B[i+1] = C[i] + D[i];
      A[i+1] = A[i+1] + B[i+1];
 }
 B[101] = C[100] + D[100];
```



LLP

for(i=2;i<=100;i++) {

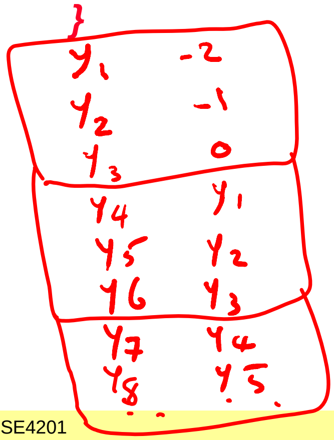
y[i]=y[i-1]+y[i]

}

for(i=2;i<=100;i++) {

y[i]=y[i-3]+y[i]

}



Finding Dependences

- Finding dependences in the program is very important for renaming and executing instructions in parallel.
- Arrays and pointers makes finding dependences very difficult.
- Assume array indices are *affine*, which means on the form $a \times i + b$ where a and b are constant.
- GCD test can be used to detect dependences.

GCD Test

- Assume we stored an array with index value of $a \times i + b$ and loaded an array with an index value of $c \times j + d$
- Are they pointing to the same location?
- Assume the loop limit is *m,n*
- Are there

$$j,k \quad m \le j,k \le n \text{ such that } a \times j + b = c \times k + d$$

GCD Test

- A simple and **sufficient** test for absence can be found.
- If a loop dependence exists, then

$$GCD(c,a)\text{ must divides }(d-b)$$

it divides

- If that test fails, there is no guarantee there is dependence (loop bound)

GCD Test

```
for(i=1; i<=100; i=i+1) {
                x[2*i+3] = x[2*i] * 5.0;
          }
```

a = 2 b = 3 c = 2 d = 0

GCD(a, c) = 2

d - b = -3

2 does not divide -3 ⇒ No dependence is not possible.

5,7,9,11,13,15,17,19,21,23,….

4,6,8,10,12,14,16,18,20,22,…..

Dependence Analysis -- Difficulties

- Dependence analysis is a very important tool for exploiting LLP, it can not be used in these situations
- Objects are referenced using pointers
- Array indexing using another array A[b[I]]
- Dependence may exist for some values of input, but in reality the input never takes these values.
- When we want to more than the possibility of dependence (which write causes it?)
- Dependence analysis across procedure boundaries

Dependence Analysis -- Difficulties

- **Sometimes, *points-to* analysis might help.**
- **We might be able to answer *simpler* questions, or get some hints.**
- **Do 2 pointers point to the same list?**
- **Type information**
- **Information derived when the object was allocated**
- **Pointer assignments**

Software Pipelinine

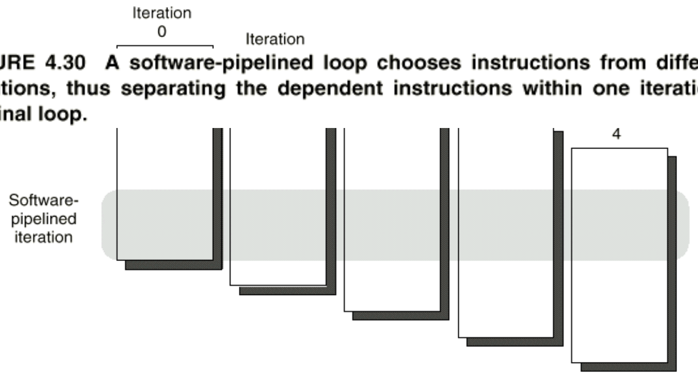


FIGURE 4.30 A software-pipelined loop chooses instructions from different loop iterations, thus separating the dependent instructions within one iteration of the original loop.

Software pipeline

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

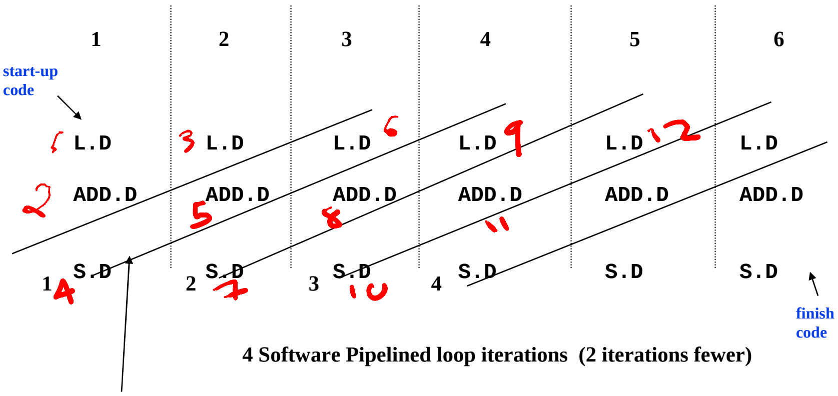
Before: Unrolled 3 times

```
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 Version

```
     L.D      F0,0(R1)
     ADD.D    F4,F0,F2
     L.D      F0,-8(R1)
1    S.D      F4,0(R1)   ;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    DADDUI   R1,R1,#-8
5    BNE      R1,R2,LOOP
     S.D      F4, 0(R1)
     ADDD     F4,F0,F2
     S.D      F4,-8(R1)
```

Software pipeline



start-up code

finish code

4 Software Pipelined loop iterations (2 iterations fewer)

Loop Body of software Pipelined Version