

CSEN
702

Microprocessors

Lecture
8/9

Instruction Level Parallelism (parts 1 and 2)

Reading

Textbook Chapter 3, Appendix H

Instruction level parallelism

- Pipelining can be considered as a form of instruction level parallelism
 - Achieved by having multiple instructions executing in the data path at the same time.
- More techniques exist for extending the basic pipelining concepts by increasing the amount of parallelism exploited among instructions.

ILP categories

- There are two largely separable approaches to exploiting ILP:
- (1) An approach that relies on hardware to help discover and exploit the parallelism **dynamically**
- (2) An approach that relies on software technology to find parallelism **statically** at compile time.

Dynamic versus static

- Processors using the dynamic, hardware-based approach, including the Intel Core series, dominate in the desktop and server markets.
- In the personal mobile device market, where energy efficiency is often the key objective, designers exploit lower levels of instruction-level parallelism.
- Thus, in 2011, most processors for the PMD market use static approaches, e.g. ARM Cortex-A8. The new ARM Cortex-A9 uses dynamic approaches.

ILP limitations

- Limitations have led to the development of multicore designs
- ILP and thread-level-parallelism should be balanced.

CPI (revisited again)

$$\text{Pipeline CPI} = \text{Ideal pipeline CPI} + \text{Structural stalls} + \text{Data hazard stalls} + \text{Control stalls}$$

* By reducing each of the terms of the right-hand side, we decrease the overall pipeline CPI or, alternatively, increase the IPC (instructions per clock)

Technique	Reduces
Forwarding and bypassing	Potential data hazard stalls
Delayed branches and simple branch scheduling	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences
Loop unrolling	Control hazard stalls
Branch prediction	Control stalls
Dynamic scheduling with renaming	Stalls from data hazards, output dependences, and antidependences
Hardware speculation	Data hazard and control hazard stalls
Dynamic memory disambiguation	Data hazard stalls with memory
Issuing multiple instructions per cycle	Ideal CPI
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls

Loop level parallelism

- The simplest and most common way to increase the ILP is to exploit parallelism among iterations of a loop.
- This type of parallelism is often called loop-level parallelism.
- **Example:** loop that adds two 1000-element arrays and is completely parallel:

Example A

```
for (i=0; i<=999; i=i+1)
    x[i] = x[i] + y[i];
```

Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little or no opportunity for overlap.

Loop level parallelism

- We should use techniques to convert this loop level parallelism into Instruction level parallelism.
 - Static Loop unrolling (by the compiler)
 - Dynamic loop unrolling (by the hardware)
 - SIMD (Single instruction, Multiple data) architectures
 - Vector processors

Data dependency

- An instruction j is data dependent on instruction i if either of the following holds:
 - ✓ • 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 . (transitional)
- Instruction like ADD R1,R1,R1 is not a problem. (no dependency)

Example A

```
Loop:    L.D    F0,0(R1)    ;F0=array element
         ADD.D  F4,F0,F2    ;add scalar in F2
         S.D    F4,0(R1)    ;store result
         DADDUI R1,R1,#-8    ;decrement pointer 8 bytes
         BNE    R1,R2,LOOP  ;branch R1!=R2
```

How to improve or tackle this?

- **How to overcome limitations of data dependency?**

- A dependence can be overcome in two different ways:

- ✓ 1. Maintaining the dependence but avoiding a hazard,
- ✓ 2. Eliminating a dependence by transforming the code.
 - Scheduling the code is the primary method used to avoid a hazard without altering a dependence, and such scheduling can be done both by the compiler and by the hardware.

How to improve or tackle this?

- When the data flow occurs in a **register**, detecting the dependence is straightforward since the register names are fixed in the instructions.
- Dependences that flow through **memory** locations are more difficult to detect.
 - Since two addresses may refer to the same location but look different: For example, 100(R4) and 20(R6) may be identical memory addresses

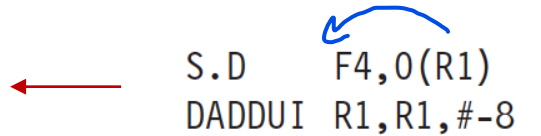
Name dependency

- A name dependence occurs when two instructions use the same register or memory location, called a name , but there is no flow of data between the instructions associated with that name.

- Two types of name dependence:

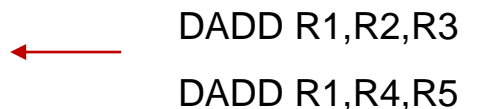
i. **An anti-dependence** between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. The original ordering must be preserved to ensure that i reads the correct value.

S.D F4,0(R1)
DADDUI R1,R1,#-8



ii. **An output dependence** occurs when instruction i and instruction j write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j .

DADD R1,R2,R3
DADD R1,R4,R5



Name dependency: Renaming

- Because a name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict.
- This renaming can be more easily done for register operands, where it is called register renaming.
- Register renaming can be done either statically by a compiler or dynamically by the hardware.

Data hazards types

- RAW (read after write)

instruction i

instruction j

- Instruction j tries to read a source before i writes it, so j incorrectly gets the old value.
- This hazard is the most common type and corresponds to a true data dependence.

Data hazards types

- **WAW (write after write)**
 - j tries to write an operand before it is written by i.
- The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination.
- This hazard corresponds to an output dependence.
- WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.

Data hazards types

- **WAR (write after read)**

- j tries to write a destination before it is read by i, so i incorrectly gets the new value.

- This hazard arises from an anti-dependence (or name dependence).

- This is usually not a problem in pipelines because reading always happens at the decode stage and writing at the end.

- **BUT** the problem might arise when we re-order the instructions!

Control dependency

```
if p1 {  
    S1;  
};  
if p2 {  
    S2;  
}
```

1. An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.
 - For example, we cannot take an instruction from the then portion of an if statement and move it before the if statement.
2. An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch.
 - For example, we cannot take a statement before the if statement and move it into the then portion.

Control dependency

- Exceptions and data flow must be preserved after instruction re-ordering.

	DADDU	R2,R3,R4
	BEQZ	R2,L1
L1:	LW	R1,0(R2)

- Is there any data dependency between LW and BEQZ?
- Can we interchange them?

	DADDU	R1,R2,R3
	BEQZ	R4,L
	DSUBU	R1,R5,R6
L:	...	
	OR	R7,R1,R8

- Can we move the OR before the branch?

Part 2

Compiler techniques for ILP

Loop scheduling and unrolling

- 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, the execution of 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

Loop scheduling and unrolling

- For the following slide, we will assume the following latencies for 2 consecutive instructions.

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

- Number of intervening clock cycles needed to avoid a stall.
- If not given, you can always compute them

- Full forwarding** is applied.
- Branches outcomes are computed at the decode stage.
- All units are fully pipelined.

Loop scheduling and unrolling

Example B

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

This loop is parallel by noticing that the body of each iteration is independent.

Loop:	L.D	F0,0(R1)	;F0=array element
	ADD.D	F4,F0,F2	;add scalar in F2
	S.D	F4,0(R1)	;store result
	DADDUI	R1,R1,#-8	;decrement pointer
			;8 bytes (per DW)
	BNE	R1,R2,Loop	;branch R1!=R2

Notes:

- Highest address x[999] is in R1
- R2 contains Lowest address x[0] minus 8.
- R2 is pre-computed.

Loop scheduling

- Let's analyze it:

Example B

```
Loop:   L.D      F0,0(R1)      ;F0=array element
        ADD.D    F4,F0,F2      ;add scalar in F2
        S.D      F4,0(R1)      ;store result
        DADDUI   R1,R1,#-8      ;decrement pointer
                                   ;8 bytes (per DW)
        BNE      R1,R2,Loop     ;branch R1!=R2
```

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

without any scheduling

			<u>Clock cycle issued</u>
			↓
Loop:	L.D	F0,0(R1)	1
	<i>stall</i>		2
	ADD.D	F4,F0,F2	3
	<i>stall</i>		4
	<i>stall</i>		5
	S.D	F4,0(R1)	6
	DADDUI	R1,R1,#-8	7
	<i>stall</i>		8
←	BNE	R1,R2,Loop	9

Total: 9 cycles / iteration

Loop scheduling (cont.)

Example B

Loop: L.D F0,0(R1) ;F0=array element
ADD.D F4,F0,F2 ;add scalar in F2
S.D F4,0(R1) ;store result
DADDUI R1,R1,#-8 ;decrement pointer
;8 bytes (per DW)
BNE R1,R2,Loop ;branch R1!=R2

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

With scheduling

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

Total: 7 cycles / iteration 24

Loop scheduling (cont.)

Example B

```
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
```

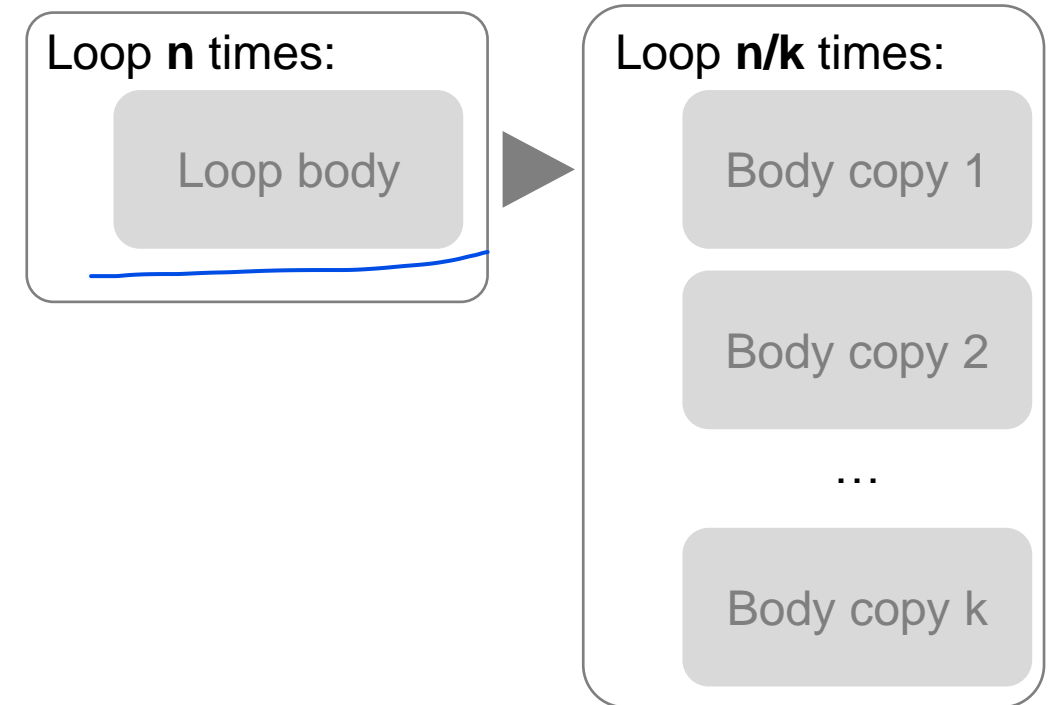
- we complete one loop iteration and store back one array element every seven clock cycles, but the actual work of operating on the array element takes just three (the load, add, and store) of those seven clock cycles.
- The remaining four clock cycles consist of **loop overhead**: the DADDUI and BNE and two stalls

• Improvement:

- 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, adjusting the loop termination code.

Loop unrolling

- Assume the loop is originally n iterations, unroll the loop k times (copy and paste the loop body k times)
 - \rightarrow then you only have to loop (n/k) times.
- Usually, avoid register re-use during unrolling.
- All initial values (loop variables) must be also taken care of.



Loop unrolling

Example B

Original code

```
Loop:  L.D      F0,0(R1)      ;F0=array element
        ADD.D   F4,F0,F2      ;add scalar in F2
        S.D     F4,0(R1)      ;store result
        DADDUI  R1,R1,#-8      ;decrement pointer
                                   ;8 bytes (per DW)
        BNE     R1,R2,Loop     ;branch R1!=R2
```

Notes:

- Highest address x[999] is still in R1
- R2 is pre-computed to be (lowest address - 8)

Basic unrolling 4 times

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

Loop unrolling

- Let's analyze the stalls

Loop:	L.D	F0, 0(R1)	1 stall
	ADD.D	F4, F0, F2	2 stalls
	S.D	F4, 0(R1)	
	L.D	F6, -8(R1)	1 stall
	ADD.D	F8, F6, F2	2 stalls
	S.D	F8, -8(R1)	
	L.D	F10, -16(R1)	1 stall
	ADD.D	F12, F10, F2	2 stalls
	S.D	F12, -16(R1)	
	L.D	F14, -24(R1)	1 stall
	ADD.D	F16, F14, F2	2 stalls
	S.D	F16, -24(R1)	
	DADDUI	R1, R1, #-32	1 stall
	BNE	R1, R2, Loop	

TOTAL:

14 cycles for instructions + 13 stalls = 27 cycles

On average, each iteration (here we have 4 iterations):

→ $27/4 = 6.75$ cycles.

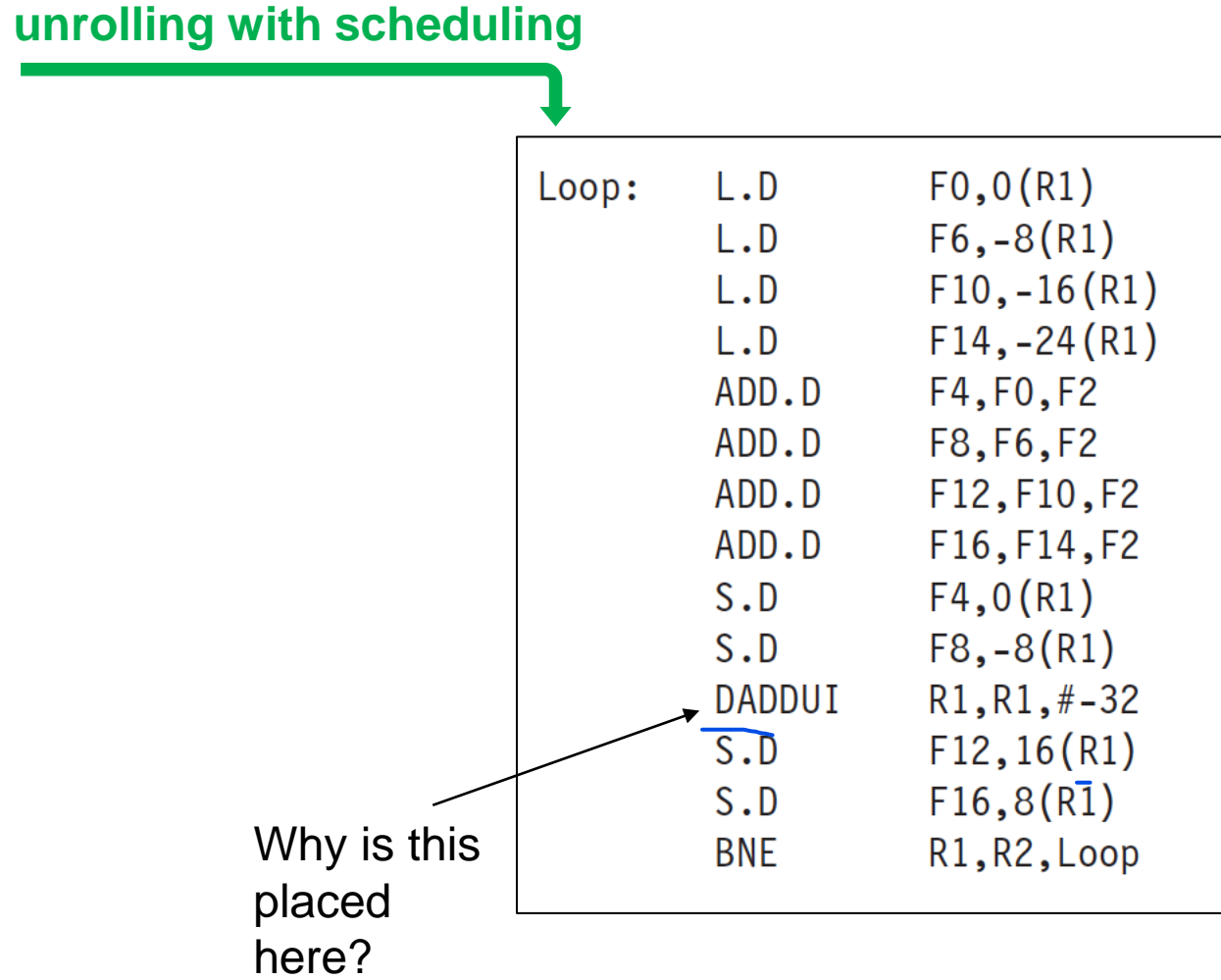
→ This is slightly better than 7 cycles obtained with loop scheduling without unrolling.

Can we do better?

Loop unrolling **with** scheduling

```
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
```

unrolling with 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
```

Why is this
placed
here?

TOTAL:
14 cycles for
instructions + **zero
stalls** = 14 cycles

On average, each
iteration (here we
have 4 iterations):
→ $14/4 = 3.5$ cycles.

→ This is much better
than 7 cycles
obtained with loop
scheduling only or
6.75 obtained with
loop unrolling only

Limitations

- **Unrolling results in the growth in code size.**
 - For larger loops, the code size growth may be a concern particularly if it causes an increase in the instruction cache **miss rate**.
- Another factor often more important than code size is the potential shortfall in registers that is created by aggressive unrolling and scheduling: called **register pressure**.

Loop-carried dependence

- Not all loops can be made parallel
- If the values in a loop iterations depend on the results of the previous iteration(s), the dependence is called loop-carried.

Example C

```
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 */  
}
```

- In this loop, the values A[i+1] and B[i+1] depend on the previous iteration. May pose a problem for unrolling!
- Another dependence is in statement S2, where B[i+1] depends on the previous statement S1, within the same loop. That's no problem for loop unrolling.

Loop carried dependence

- A loop is parallel if it can be written without a cycle in the dependences, since the absence of a cycle means that the dependences give a partial ordering on the statements.

Example D

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

i=1

A[1]=A[1] + B[1]

B[2]=C[1] + D[1]

i=2

A[2]=A[2] + B[2]

B[3]=C[2] + D[2]

i=3

A[3]=A[3] + B[3]

B[4]=C[3] + D[3]

.

.

.

**Loop-carried
dependence**

Loop carried dependence

Can we just interchange the statements?

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

i=1

B[2]=C[1] + D[1]

A[1]=A[1] + B[1]

i=2

B[3]=C[2] + D[2]

A[2]=A[2] + B[2]

i=3

B[4]=C[3] + D[3]

A[3]=A[3] + B[3]

.

.

.

Example D (cont.):

Example D

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

i=1

A[1]=A[1] + B[1]

B[2]=C[1] + D[1]

i=2

A[2]=A[2] + B[2]

B[3]=C[2] + D[2]

i=3

A[3]=A[3] + B[3]

B[4]=C[3] + D[3]

i=100

A[100]=A[100] + B[100]

B[101]=C[100] + D[100]

Solution:

- Pack the dependent statements into 1 iteration.
- Hanging statements (first one in first iteration, and last one in last iteration need to be executed outside the 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];

Result

Example D new code

```
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];
```

A[1]=A[1] + B[1]

i=1

B[2]=C[1] + D[1]

A[2]=A[2] + B[2]

i=2

B[3]=C[2] + D[2]

A[3]=A[3] + B[3]

i=3

B[4]=C[3] + D[3]

A[4]=A[4] + B[4];

...

...

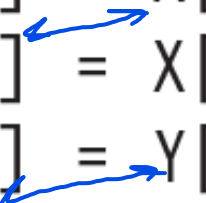
B[101]=C[100] + D[100]

Technique 2: Name changing

- The compiler may resort to name changing to remove dependencies such as anti-dependence and output dependence.

Example E

```
for (i=1; i<=100; i=i+1) {  
  — Y[i] = X[i] / c;  
    X[i] = X[i] + c;  
  — Z[i] = Y[i] + c;  
    Y[i] = c - Y[i];  
}
```



What are the type dependencies found in this code?

Technique 2: Name changing

Example E: Finding dependencies types

```
for (i=1; i<=100; i=i+1) {  
    Y[i] = X[i] / c; /* S1 */  
    X[i] = X[i] + c; /* S2 */  
    Z[i] = Y[i] + c; /* S3 */  
    Y[i] = c - Y[i]; /* S4 */  
}
```

1. There are true dependences from S1 to S3 and from S1 to S4 because of Y[i]. These are not loop carried, so they do not prevent the loop from being considered parallel. These dependences will force S3 and S4 to wait for S1 to complete.
2. There is an anti-dependence from S1 to S2, based on X[i].
3. There is an anti-dependence from S3 to S4 for Y[i].
4. There is an output dependence from S1 to S4, based on Y[i].

Technique 2: Name changing

Example E: Solution

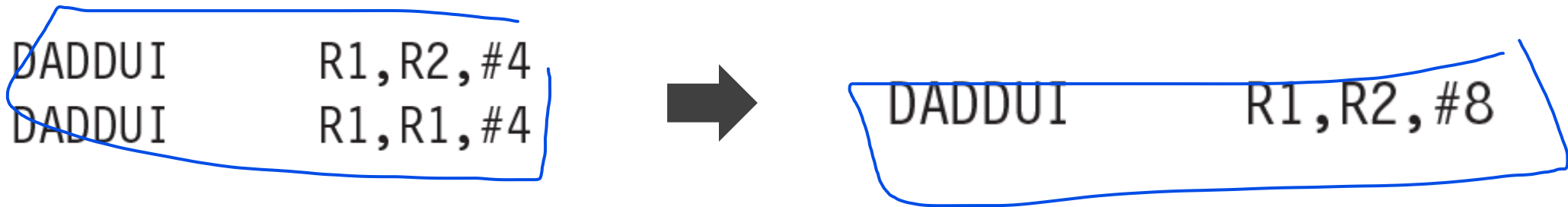
```
for (i=1; i<=100; i=i+1) {  
    Y[i] = X[i] / c; /* S1 */  
    X[i] = X[i] + c; /* S2 */  
    Z[i] = Y[i] + c; /* S3 */  
    Y[i] = c - Y[i]; /* S4 */  
}  
→  
for (i=1; i<=100; i=i+1 {  
    /* Y renamed to T to remove output dependence */  
    T[i] = X[i] / c;  
    /* X renamed to X1 to remove antidependence */  
    X1[i] = X[i] + c;  
    /* Y renamed to T to remove antidependence */  
    Z[i] = T[i] + c;  
    Y[i] = c - T[i];  
}
```



After the loop, the variable X has been renamed X1. In code that follows the loop, the compiler can simply replace the name X by X1

Technique 3: Copy propagation

- Simple arithmetic optimizations to reduce dependencies.



- We actually used this type of optimization in loop unrolling.

Technique 4: tree height reduction

- Arithmetic optimizations to reduce “height” of a tree structure that represents a computation.



$$\begin{aligned} R8 &= R4 + R7 \\ &= (R1 + R6) + R7 \\ &= ((R2 + R3) + R6) + R7 \\ &= (R2+R3) + (R6+R7) \end{aligned}$$