Exploiting Instruction Level Parallelism
CS 740
Oct. 1, 1997

Code Optimizations

- Scheduling
- Loop unrolling
- Software pipelining

Compilation Techniques

- Dependency graphs
- Dependency analysis

Multicycle Operations

MIPS Instructions Requiring > 1 Cycle

`lw, lwc1`	2	(Load delay slot)
`mult, multu`	~12	
`div, divu`	~32	
`add.s, add.d`	2	
`mul.s, mul.d`	4–5	
`div.s, div.d`	~8–10	

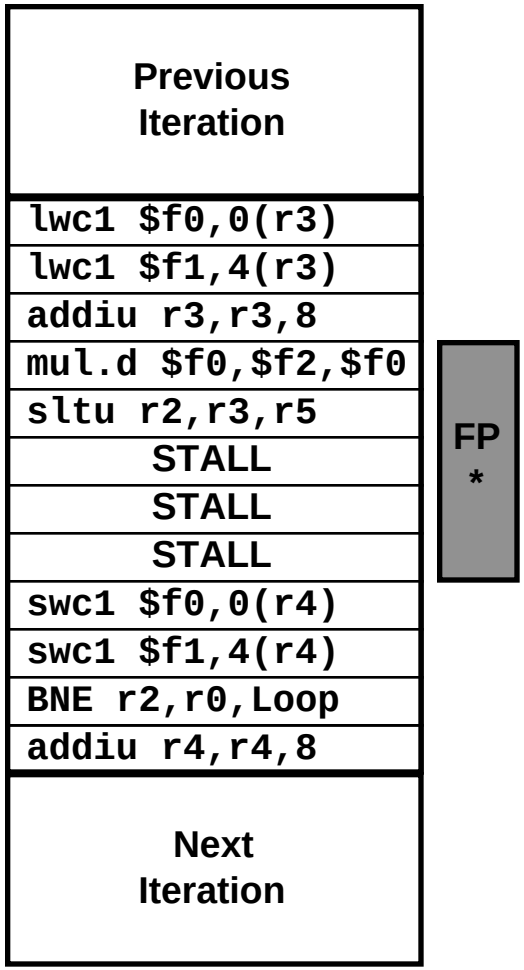
Operation (except for `lw`)

- **Instruction initiates multicycle operation**
- **Successive operations can potentially execute without delay**
  - As long as don't require result of multicycle operation
  - And don't require same hardware resources
- **If run into problem, stall processor until operation completed**

Code Scheduling Example

- **Multiply elements of vector b by scalar c and store in vector a**
- **Compile into 9 instructions**
- **Requires 12 cycles**
  - 3 cycle stall from FP multiply

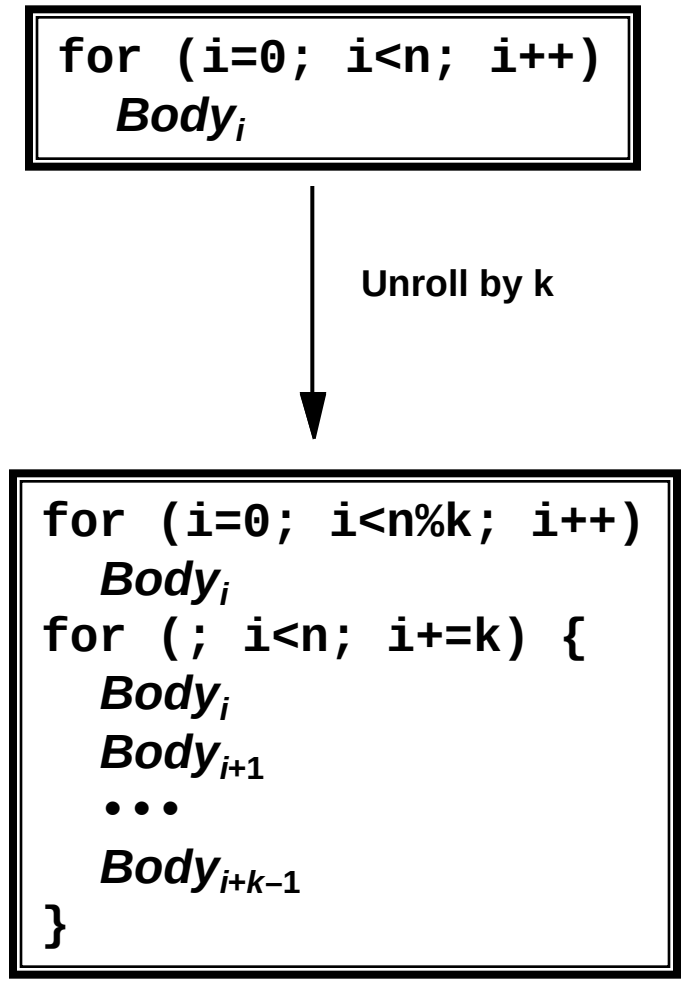
```
#define CNT 256
static double a[CNT], b[CNT];
static double c;

void loop1(void)
{
  double *anext = a;
  double *bnext = b;
  double *bdone = b+CNT;
  double tc = c;
  while (bnext < bdone)
    *anext++ = *bnext++ * tc;
}
```



Loop Unrolling

Advanced Optimization

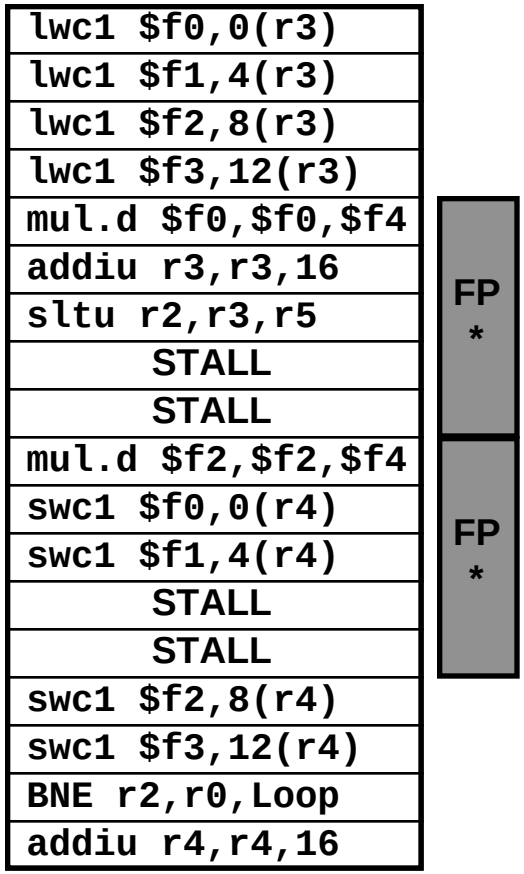
- **Combine loop iterations**
- **Reduce loop overhead**
- **Expose optimizations across iterations**
  - E.g., common subexpressions
- **More opportunities for clever scheduling**



Loop Unrolling Example

- **Unrolling by 2 gives 18 cycle loop**
  - 9 cycles / element
- **4 overhead operations spread over 2 elements**
- **1 less stall cycle / element**

```
void loop2(void)
{
  double *anext = a;
  double *bnext = b;
  double *bdone = b+CNT;
  double tc = c;
  while (bnext < bdone) {
    double b0 = bnext[0];
    double b1 = bnext[1];
    bnext += 2;
    anext[0] = b0 * tc;
    anext[1] = b1 * tc;
    anext += 2;
  }
}
```



Software Pipelining

Advanced Optimization

- Spread code for single iteration over multiple loops
- Tends to stretch out data dependent operations
- Allows more effective code scheduling

```
for (i=0; i<n; i++) {
  A_i ; B_i ; C_i
}
```

3-way pipeline

```
A_0 ; B_0 ; A_1
for (i=2; i<n; i++) {
  C_i-2 ; B_i-1 ; A_i
}
C_n-2 ; B_n-1 ; C_n-1 ;
```

Software Pipelining Example

```
void loop1(void)
{
  double *anext = a;
  double *bnext = b;
  double *bdone = b+CNT;
  double tc = c;
  while (bnext < bdone) {
    double load = *bnext++;
    double prod = load * tc;
    *anext++ = prod;
  }
}
```

- **Operations**
  - A: load element from b
  - B: multiply by c
  - C: store in a
- **9 cycles / iteration**
  - No stalls!

3-Way Pipelined

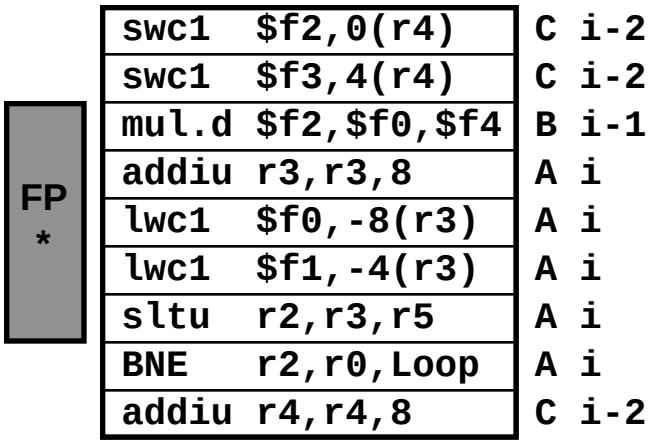
```
void pipe(void)
{
  double tc = c;
  /* Prologue */
  double prod = b[0] * tc; /* A0, B0 */
  double load = b[1];      /* A1 */
  double *anext = a;
  double *bnext = b+2;
  double *bdone = b+CNT;
  /* Loop */
  while (bnext < bdone) {
    *anext++ = prod;       /* Ci-2 */
    prod = load * tc;      /* Bi-1 */
    load = *bnext++;       /* Ai */
  }
  /* Epilogue */
  a[CNT-2] = prod;         /* Cn-2 */
  a[CNT-1] = load * tc;    /* Bn-1, Cn-1 */
}
```

Software Pipelined Loop

Loop

```
while (bnext < bdone) {
    *anext++ = prod;  /* Ci-2 */
    prod = load * tc; /* Bi-1 */
    load = *bnext++;  /* Ai */
}
```

Generated Code



- **Operations**
  - A: load element from b
  - B: multiply by c
  - C: store in a
- **9 cycles / iteration**
  - No stalls!

Tracing Single Iteration

Loop

```
while (bnext < bdone) {
  *anext++ = prod;  /* Ci-2 */
  prod = load * tc; /* Bi-1 */
  load = *bnext++;  /* Ai */
}
```

- **Operations**
  - A: load element from b
  - B: multiply by c
  - C: store in a
- **9 cycles / iteration**
  - No stalls!

swc1	$f2,0(r4)	C i-2
swc1	$f3,4(r4)	C i-2
mul.d	$f2,$f0,$f4	B i-1
addiu	r3,r3,8	A i
lwc1	$f0,-8(r3)	A i
lwc1	$f1,-4(r3)	A i
sltu	r2,r3,r5	A i
BNE	r2,r0,Loop	A i
addiu	r4,r4,8	C i-2
swc1	$f2,0(r4)	C i-1
swc1	$f3,4(r4)	C i-1
mul.d	$f2,$f0,$f4	B i
addiu	r3,r3,8	A i+1
lwc1	$f0,-8(r3)	A i+1
lwc1	$f1,-4(r3)	A i+1
sltu	r2,r3,r5	A i+1
BNE	r2,r0,Loop	A i+1
addiu	r4,r4,8	C i-1
swc1	$f2,0(r4)	C i
swc1	$f3,4(r4)	C i
mul.d	$f2,$f0,$f4	B i+1
addiu	r3,r3,8	A i+2
lwc1	$f0,-8(r3)	A i+2
lwc1	$f1,-4(r3)	A i+2
sltu	r2,r3,r5	A i+2
BNE	r2,r0,Loop	A i+2
addiu	r4,r4,8	C i

Optimization Results (Clocks/Element)

Transformation	MIPS	PPC 604e	Pentium II
Straight Loop	12	4	9
Unroll 2X	9	3	8
Unroll 4X	8	3	7.75
Pipeline	9	3	9.5
Pipeline & Unroll 2X	7.5	2.5	8
Theoretical Limit	5	2	2?

- **MIPS: Can keep unrolling to reduce loop overhead**
  - Left with 2 **`lwc1`**'s, 1 **`mul.d`**, and 2 **`swc1`**'s
- **604: Single Load/Store unit**
- **Pentium II: FP Multiply has throughput of 1 Mult / 2 cycles**

604e Code Example

- **Unrolling by 2 gives 6 cycle loop**
  - 3 cycles / element

Loop2 inner loop

```
while (bnext < bdone) {
  double b0 = bnext[0];
  double b1 = bnext[1];
  bnext += 2;
  anext[0] = b0 * tc;
  anext[1] = b1 * tc;
  anext += 2;
}
```

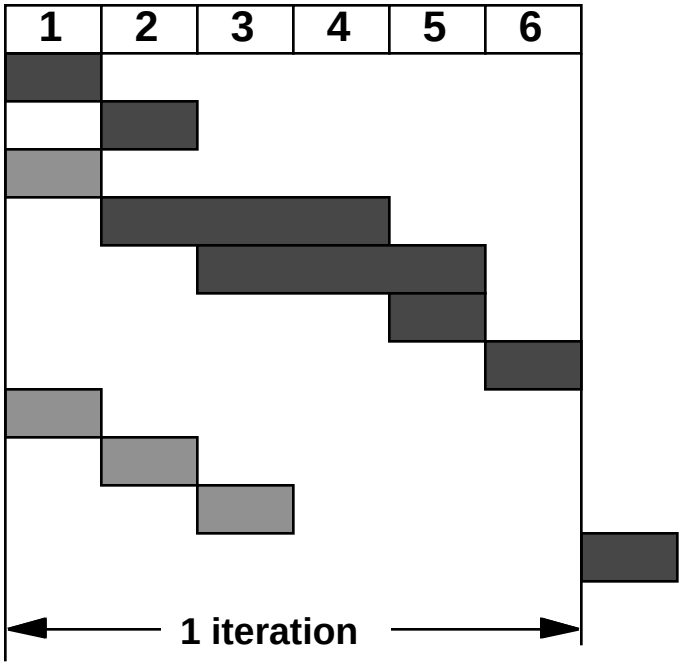
Generated Code

```
lfd     fp0,0(r5)    # bnext[0]
lfd     fp3,8(r5)    # bnext[1]
addi    r5,r5,16     # bnext += 2
fmul    fp1,fp0,fp2  # b0 * tc
fmul    fp0,fp3,fp2  # b1 * tc
stfd    fp1,0(r4)    # anext[0]
stfd    fp0,8(r4)    # anext[1]
addi    r4,r4,16     # anext += 2
cmplw   r5,r0        # bnext < bdone
blt     *-36         # Loop
```

604e Code Schedule

- **Single Load / Store Unit**
  - Memory operations in program order
- **FP Multiply has 3 cycle latency**
  - But fully pipelined
- **Branch prediction eliminates any branch penalty**

```
lfd   fp0,0(r5)
lfd   fp3,8(r5)
addi  r5,r5,16
fmul  fp1,fp0,fp2
fmul  fp0,fp3,fp2
stfd  fp1,0(r4)
stfd  fp1,8(r4)
addi  r4,r4,16
cmplw r5,r0
blt   *-36
lfd   fp0,0(r5)
      . . .
```



Data dependences

Data dependence (true or flow dependence)

- **There is a data dependence between S1 and S2 if S1 produces a result that is consumed by S2.**
- **Example:**

```
S1:     addiu r1, r2,  r3
S2:     addiu r4, r4, r1
```

- **Example:**

```
S1:     a[1] = 3;
S2:     a[2] = a[1];
```

- **Necessary condition for RAW hazards**

Name (false) dependences

Antidependence:

- **There is an antidependence between S1 and S2 if S1 reads a register or memory location that S2 later writes.**
- **Example:**

```
S1:     addiu r4, r4, r1
S2:     addiu r1, r2,  r3
```

- **Example:**

```
S1:     a[2] = a[1];
S2:     a[1] = 3;
```

- **Necessary condition for WAR hazards.**

Name dependences (cont)

Output dependence:

- **There is an output dependence between S1 and S2 if S1 writes a register or memory location that S2 later writes.**
- **Example:**

  S1: `addiu r4, r4, r1`

  S2: `addiu r4, r5, r6`

- **Example:**

  S1: `a[1]= a[2] + 1;`

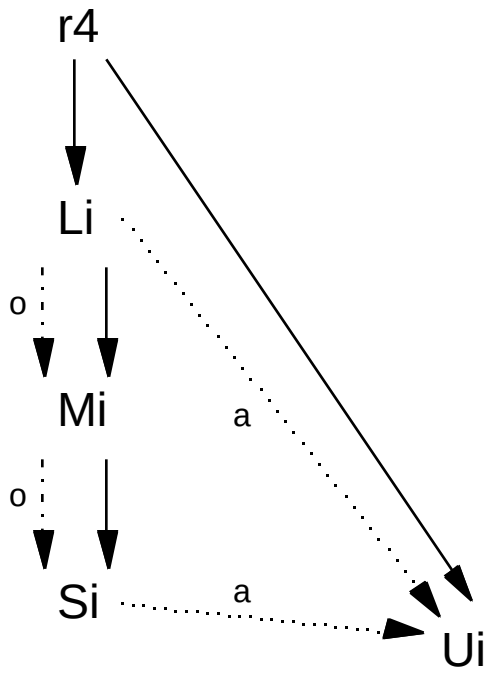
  S2: `a[1] = a[1] + 2;`

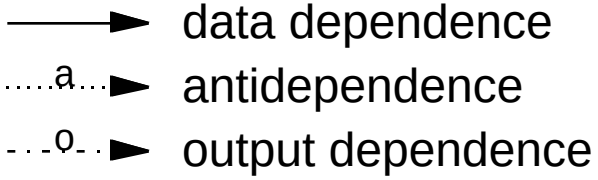
- **Necessary condition for WAW hazards.**

Dependence graphs

```
a[i] *= c;
```

```
Li:     ldc1   f0, 0(r4)
Mi:     mul.s  f0, f0, f12
Si:     swc1   f0, 0(r4)
Ui:     addiu  r4, r4, 4
```



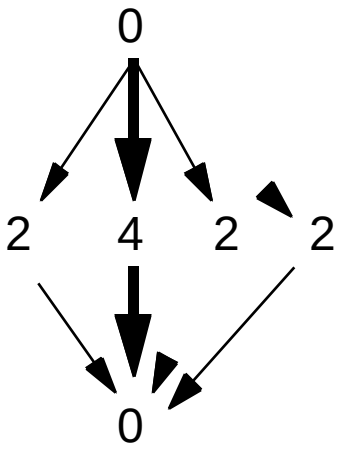


Instruction level parallelism (ILP)

Def. ILP = W/L

W **sum of the instruction node weights (execution times)**

L **critical path of node weights**



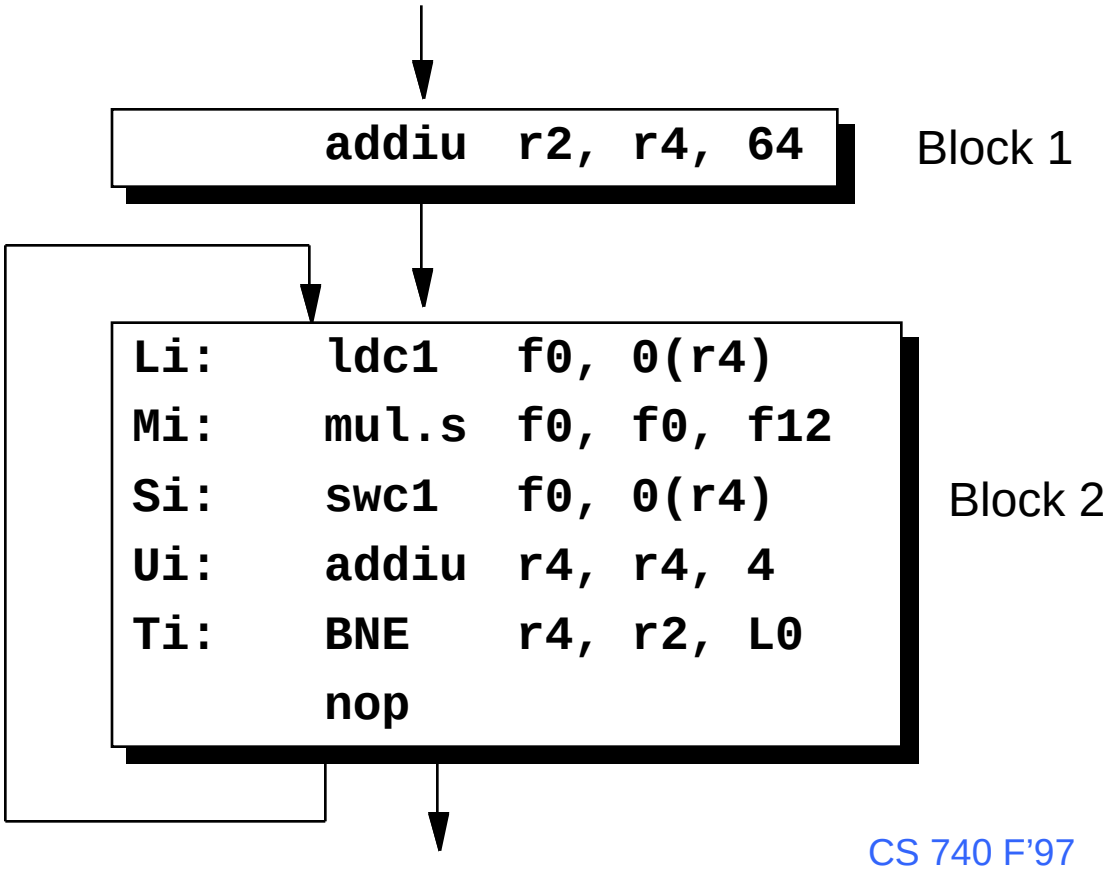
w = 10
L = 4
ILP = 2.5

w = 10
L = 10
ILP = 1

ILP is important because it characterizes the amount of freedom we have in scheduling instructions.

Basic blocks

Basic block: straightline sequence of instructions with one entry point and one exit point.

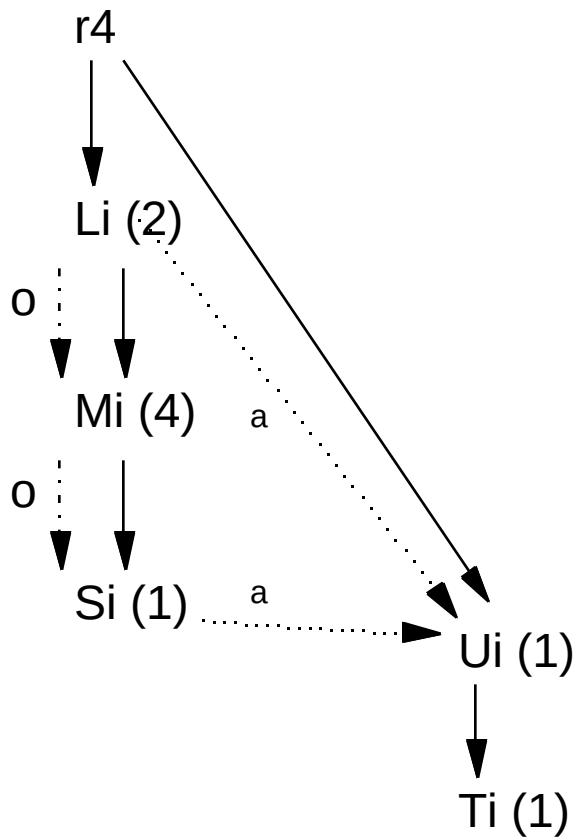


Scheduling a single basic block

```
for (i=0; i<n; i++)
    a[i] *= c;
```

Naive schedule: 10 cycles/result.

```
Li:    ldc1   f0, 0(r4)
Mi:    mul.s  f0, f0, f12
Si:    swc1   f0, 0(r4)
Ui:    addiu  r4, r4, 4
Ti:    BNE    r4, r2, Li
       nop
```

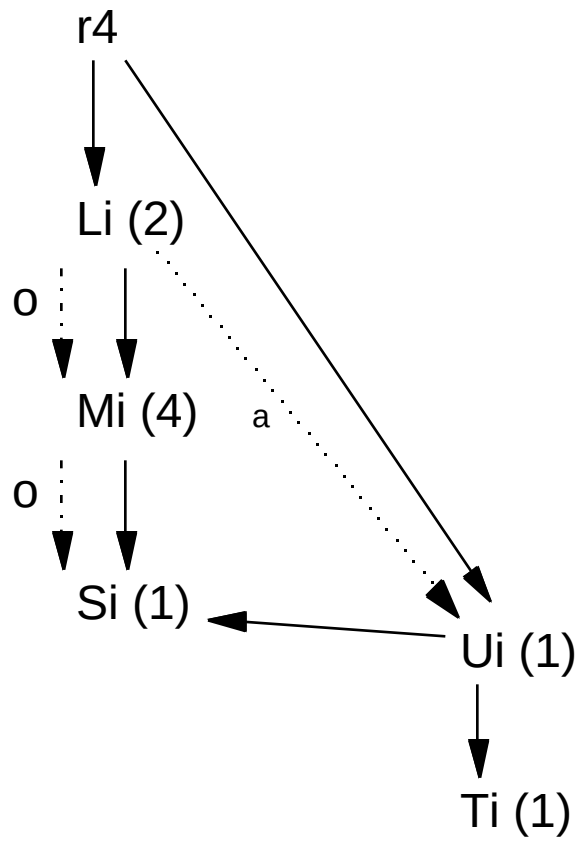


Notice that there is no ILP in this example!

Scheduling a single basic block (cont)

Optimal schedule for one basic block:
7 cycles/result.

```
Li:    ldc1   f0, 0(r4)
Mi:    mul.s  f0, f0, f12
Ui:    addiu  r4, r4, 4
Ti:    BNE    r4, r2, Li
Si:    swc1   f0, -4(r4)
```



Loop-carried dependences

Def. A dependence between an instance of a statement S in one loop iteration and an instance of S in another iteration is called a *loop carried dependence*.

No loop carried dependence:

```
for (i = 1; i <= n; i++)
{
   x[i] = x[i]  * c;
}
```

Loop carried dependence:

```
for (i = 1; i <= n; i++)
{
   x(i) = x(i-1)  * c;
}
```

A loop with no loop-carried dependences is called a *parallel loop.*

Detecting loop carried dependences

```
for (i = m; i <= n; i++) {
 x[a*i + b] = x[c*i + d];
}
```

GCD test: If a loop-carried dependence exists, then GCD(c,a) must divide d-b (i.e., d-b / GCD(c,a) is a nonzero integer)

Example:

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

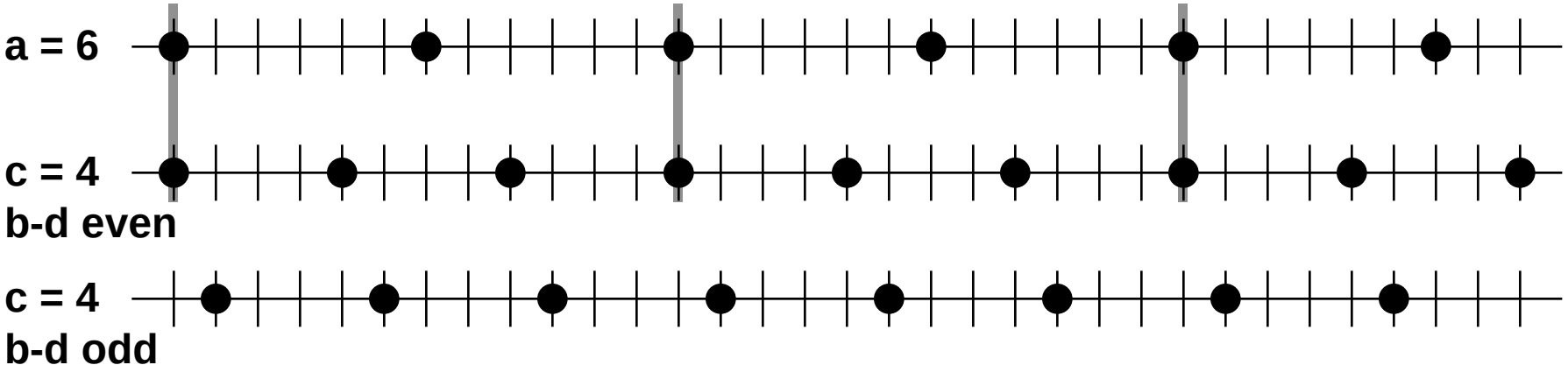
a=2, b=3, c=2, d=0
GCD(2,2) = 2, d-b = -3
-3/2 not an integer, so no loop carried dependence is possible.

Example:

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

a=1, b=0, c=1, d=0
GCD(1,1) = 1, d-b = 0
0/1 not an integer, so no loop carried dependence is possible.

Intuition Behind GCD Test



- **GCD =** p
- **Elements of form** p*i1 + o1 **and** p*i2 + o2
- **If** o1 ≠ o2 **mod** p**, then can never line up**

Detecting loop carried dependences

Example:

```
for (i =1; i <= 1; i++) {
   x(i) = x(i-1) * 5.0;
}
```

a=1, b=0, c=1, d=-1
GCD(1,1) = 1, d-b = -1
-1/1 an integer, but there is no dependence.

GCD test is sufficient for proving no dependence, but
but not necessary. In other words, the GCD test can succeed when there is no loop-carried dependence. Why?

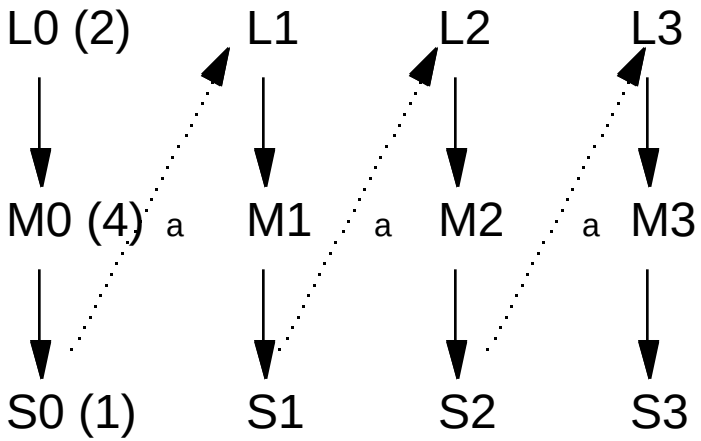
In general, dependence analysis is NP-complete, equivalent to integer programming.

Scheduling multiple basic blocks

```
L0:    ldc1   f0, 0(r4)
M0:    mul.s  f0, f0, f12
S0:    swc1   f0,0(r4)
L1:    ldc1   f0,4(r4)
M1:    mul.s  f0, f0, f12
S1:    swc1   f0,4(r4)
L2:    ldc1   f0,8(r4)
M2:    mul.s  f0, f0, f12
S2:    swc1   f0,8(r4)
L3:    ldc1   f0,12(r4)
M3:    mul.s  f0, f0, f12
U:     addiu  r4, r4,16
T:     BNE    r4, r2, L0
S3:    swc1   f0,-4(r4)
```

Unrolled loop with no register renaming:

- **7 cycles/result**
- **Antidependence inhibits ILP**

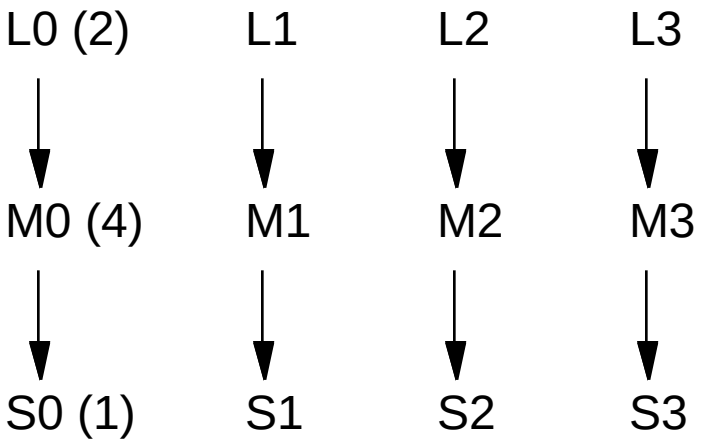


Scheduling multiple basic blocks (cont)

```
L0:    ldc1   f0, 0(r4)
M0:    mul.s  f0, f0, f12
S0:    swc1   f0,0(r4)
L1:    ldc1   f2,4(r4)
M1:    mul.s  f2, f2, f12
S1:    swc1   f2,4(r4)
L2:    ldc1   f4,8(r4)
M2:    mul.s  f4, f4, f12
S2:    swc1   f4,8(r4)
L3:    ldc1   f6,12(r4)
M3:    mul.s  f6, f6, f12
U:     addiu  r4, r4,16
T:     BNE    r4, r2, L0
S3:    swc1   f6,-4(r4)
```

Unrolled loop with register renaming:

- **7 cycles/result**
- **Constrained by instruction ordering**

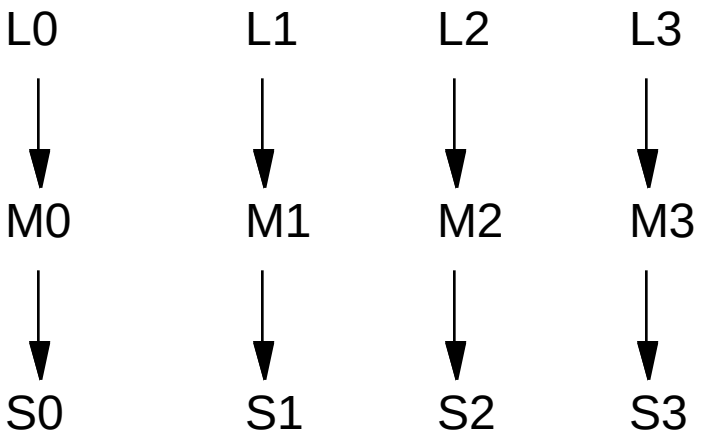


Scheduling multiple basic blocks(cont)

```
L0:    ldc1   f0, 0(r4)
L1:    ldc1   f2,4(r4)
L2:    ldc1   f4,8(r4)
L3:    ldc1   f6,12(r4)
M0:    mul.s  f0, f0, f12
M1:    mul.s  f2, f2, f12
M2:    mul.s  f4, f4, f12
M3:    mul.s  f6, f6, f12
U:     addiu  r4, r4,16
S0:    swc1   f0,-16(r4)
S1:    swc1   f2,-12(r4)
S2:    swc1   f4,-8(r4)
T:     BNE    r4, r2, L0
S3:    swc1   f6,-4(r4)
```

Unrolled loop with register renaming and improved scheduling

- **Assuming pipelined multiplier**
- **3.5 cycles/result**
- **CPI = 1**
- **Latency tolerance for mul = 5**



Observations

Optimization Critical for Low Performance Machines

- Multicycle operations give scheduling opportunity
- In-order issue requires good instruction ordering
- Need to reduce overheads due to indexing, looping

Optimization Critical for High Performance Machines

- Can exploit large amounts of ILP
- Reducing 4 cycle loop by 1 cycle yields 1.3X speed improvement
- Memory operations generally must occur in program order
- Some resources limitations

Useful Transformations

- Loop unrolling reduces loop overhead and enables more scheduling
- Software pipelining spreads apart dependent instructions