based on various sources, including the Dragon book

© Manuel Cargaleiro

Instruction Scheduling

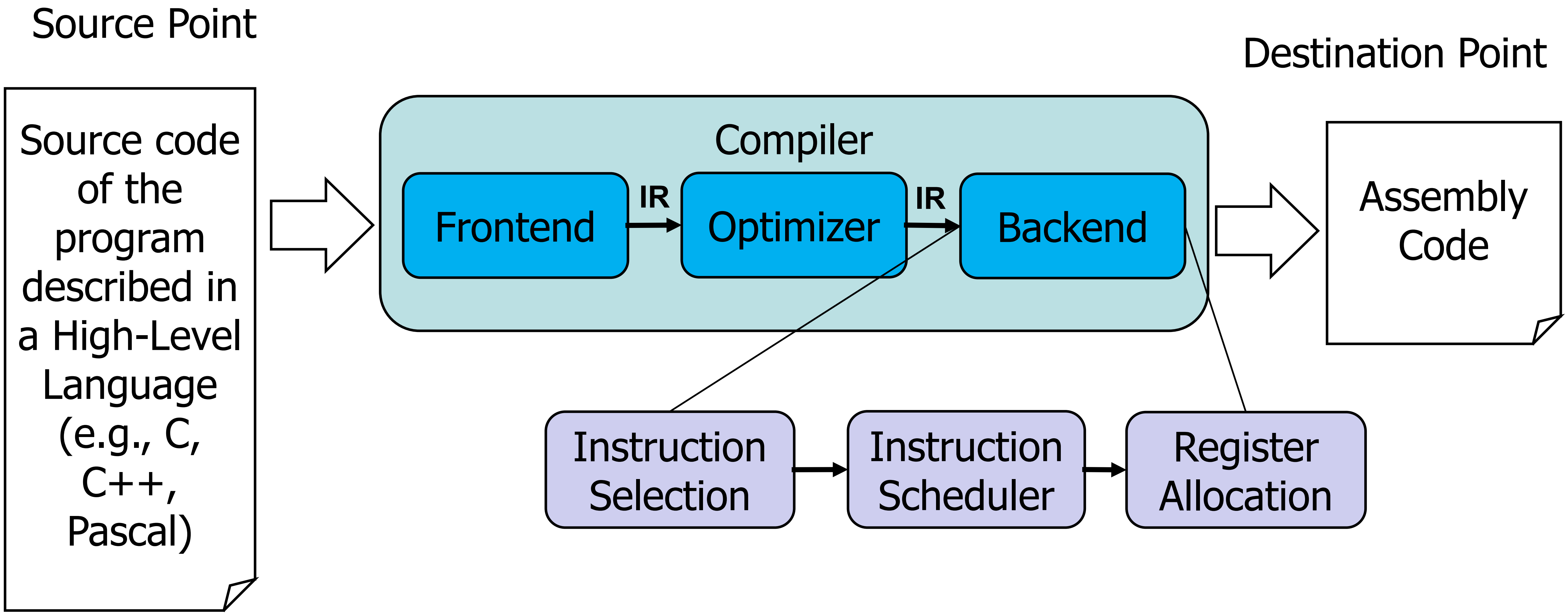
Masters in Informatics and Computing Engineering (MIEIC), 3rd Year

João M. P. Cardoso
Dep. de Engenharia Informática, Faculdade de Engenharia (FEUP), Universidade do Porto, Porto, Portugal
Email: jmpc@fe.up.pt

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Compiler Stages



Instruction Scheduling

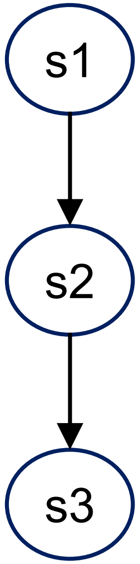
- Solving it optimally is an NP-complete problem!
- Approaches use heuristics

Data-Dependence

- Four flavors of data dependence:
  - True dependence (RAW)
  - Anti-dependence (WAR)
  - Output dependence (WAW)
  - Input dependence (RAR)
- Example:
  - s1: a =2;
  - s2: *p = 1;
  - s3: x = a;

p may point to the memory location where a is stored

Data-dependence graph (DDG)



DDG: nodes represent statements/operations, edges represent dependences

Data-Dependence Analysis

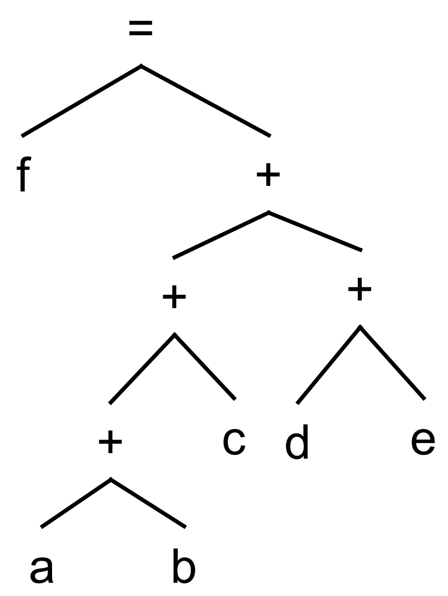
- Analysis in terms of the scope
  - Intraprocedural Analysis – at the level of the procedure being analyzed
  - Interprocedural Analysis – also considers analysis between procedures
- Types of analysis
  - Scalar variables data-dependence analysis
  - Array data-dependence analysis
  - Pointer-alias data dependence analysis
  - Reference-alias data dependence analysis

Example 1

Statement

f = (a + b) + c + (d + e)

Expression tree



Assembly Code

```
LD r1, a          // r1 = M[a]
LD r2, b          // r2 = M[b]
ADD r5, r1, r2    // r5 = r1+r2
LD r3, c          // r3 = M[c]
ADD r8, r5, r3    // r8=r5+r3
LD r4, d          // r4 = M[d]
LD r6, e          // r6 = M[e]
ADD r7, r4, r6    // r7=r4+r6
ADD r9, r8, r7    // r9=r8+r7
ST f, r9          // M[f] = r9
```

Parallel evaluation of the IR code (considering 5 functional units – FUs - and 1 clock cycle of latency for each instruction)

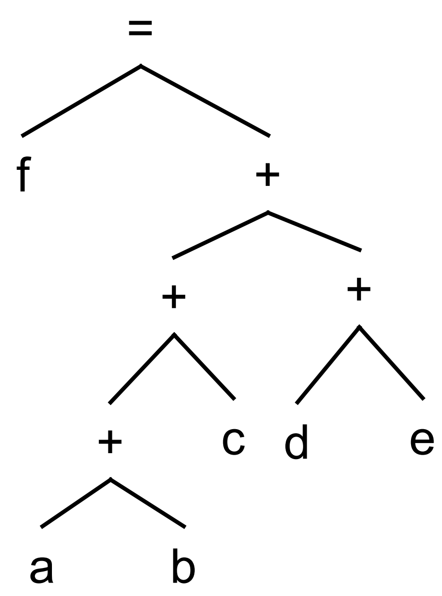
Cycles	FU1	FU2	FU3	FU4	FU5
1	LD r1, a	LD r2, b	LD r3, c	LD r4, d	LD r6, e
2	ADD r5, r1, r2	ADD r7, r4, r6			
3	ADD r8, r5, r3				
4	ADD r9, r8, r7				
5	ST f, r9				

Exercise 1

Statement

f = (a + b) + c + (d + e)

Expression tree



Assembly Code

```
LD r1, a            // r1 = M[a]
LD r2, b            // r2 = M[b]
ADD r5, r1, r2      // r5 = r1+r2
LD r3, c            // r2 = M[c]
ADD r8, r5, r3      // r8=r5+r3
LD r4, d            // r2 = M[d]
LD r6, e            // r3 = M[e]
ADD r7, r4, r6      // r7=r4+r6
ADD r9, r8, r7      // r9=r8+r7
ST f, r9            // M[f] = r9
```

Parallel evaluation of the IR code (considering 2 functional units – FUs - and 1 clock cycle of latency for each instruction)

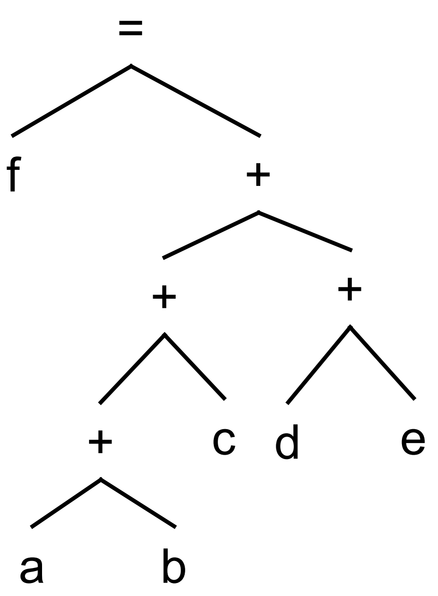
Cycles	FU1	FU2
1		
2		
3		
4		
5		
6		
7		

Exercise 1

Statement

f = (a + b) + c + (d + e)

Expression tree



Assembly Code

```
LD r1, a          // r1 = M[a]
LD r2, b          // r2 = M[b]
ADD r5, r1, r2    // r5 = r1+r2
LD r3, c          // r2 = M[c]
ADD r8, r5, r3    // r8=r5+r3
LD r4, d          // r2 = M[d]
LD r6, e          // r3 = M[e]
ADD r7, r4, r6    // r7=r4+r6
ADD r9, r8, r7    // r9=r8+r7
ST f, r9          // M[f] = r9
```

Parallel evaluation of the IR code (considering 2 functional units – FUs - and 1 clock cycle of latency for each instruction)

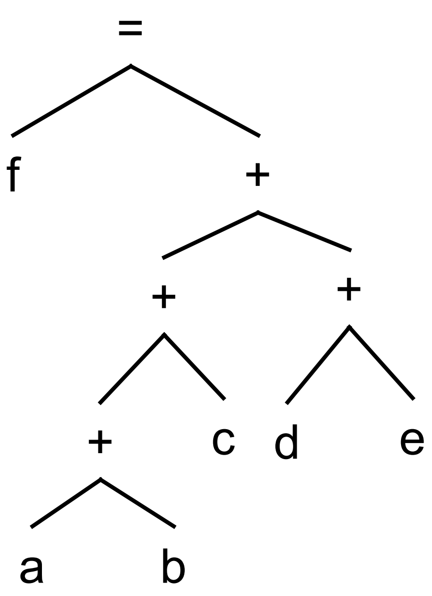
Cycles	FU1	FU2
1		
2		
3		
4		
5		
6		
7		

Exercise 1: a) – “good” priorities

Statement

f = (a + b) + c + (d + e)

Expression tree



Assembly Code

```
LD r1, a            // r1 = M[a]
LD r2, b            // r2 = M[b]
ADD r5, r1, r2      // r5 = r1+r2
LD r3, c            // r2 = M[c]
ADD r8, r5, r3      // r8=r5+r3
LD r4, d            // r2 = M[d]
LD r6, e            // r3 = M[e]
ADD r7, r4, r6      // r7=r4+r6
ADD r9, r8, r7      // r9=r8+r7
ST f, r9            // M[f] = r9
```

Parallel evaluation of the IR code (considering 2 functional units – FUs - and 1 clock cycle of latency for each instruction)

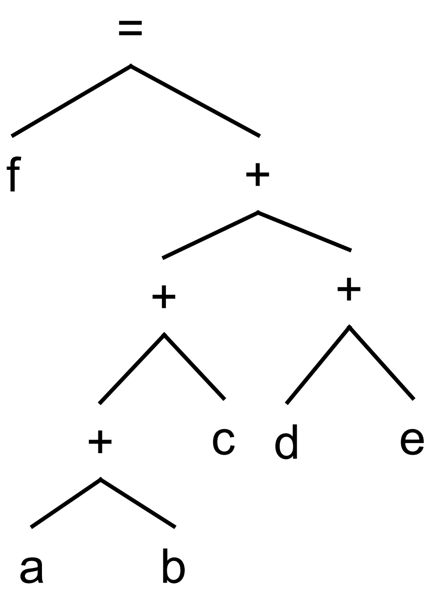
Cycles	FU1	FU2
1	LD r1, a	LD r2, b
2	ADD r5, r1, r2	LD r3, c
3	LD r4, d	LD r6, e
4	ADD r8, r5, r3	ADD r7, r4, r6
5	ADD r9, r8, r7	
6	ST f, r9	
7		

Exercise 1: b) – “bad” priorities

Statement

f = (a + b) + c + (d + e)

Expression tree



Assembly Code

```
LD r1, a          // r1 = M[a]
LD r2, b          // r2 = M[b]
ADD r5, r1, r2    // r5 = r1+r2
LD r3, c          // r2 = M[c]
ADD r8, r5, r3    // r8=r5+r3
LD r4, d          // r2 = M[d]
LD r6, e          // r3 = M[e]
ADD r7, r4, r6    // r7=r4+r6
ADD r9, r8, r7    // r9=r8+r7
ST f, r9          // M[f] = r9
```

Parallel evaluation of the IR code (considering 2 functional units – FUs - and 1 clock cycle of latency for each instruction)

Cycles	FU1	FU2
1	LD r3, c	LD r4, d
2	LD r1, a	LD r6, e
3	LD r2, b	ADD r7, r4, r6
4	ADD r5, r1, r2	
5	ADD r8, r5, r3	
6	ADD r9, r8, r7	
7	ST f, r9	

Machine Model

- Basic Machine Model, $M=\langle R,T\rangle$, consists of:
  1. A vector $R=[r_1, r_2, \ldots r_n]$ representing hardware resources, where r_i is the number of units available of the ith kind of resource. Examples of typical resource types include: memory access units, ALU's and floating-point functional units.
  2. A set of operation types T, such as loads, stores, arithmetic operations, etc.

Resource Reservation Table (RT)

- A matrix in the form of $r \times c$, where r represents the hardware resources, and c the cycles of the schedule
- Each entry in this table is an instruction that uses a resource for that cycle

Data-Dependence Graph (DDG)

- Each basic block of machine instructions is represented by a DDG, G=(N,E), having:
  - a set of nodes N representing the operations in the machine instructions in the block, and
  - a set of directed edges E representing the data-dependence constraints among operations
- The nodes and edges of G are constructed as follows:
  1. Each operation i in N has a resource-reservation table RT_i, whose value is simply the resource-reservation table associated with the operation type of i
  2. Each edge E is labeled with a delay d, indicating that the destination node must be issued no earlier than d clocks after the source node is issued.

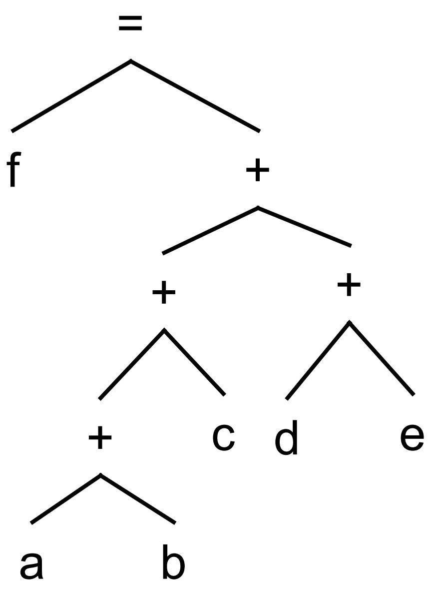
What is a basic block?

DDG for Example E1

Statement

f = (a + b) + c + (d + e)

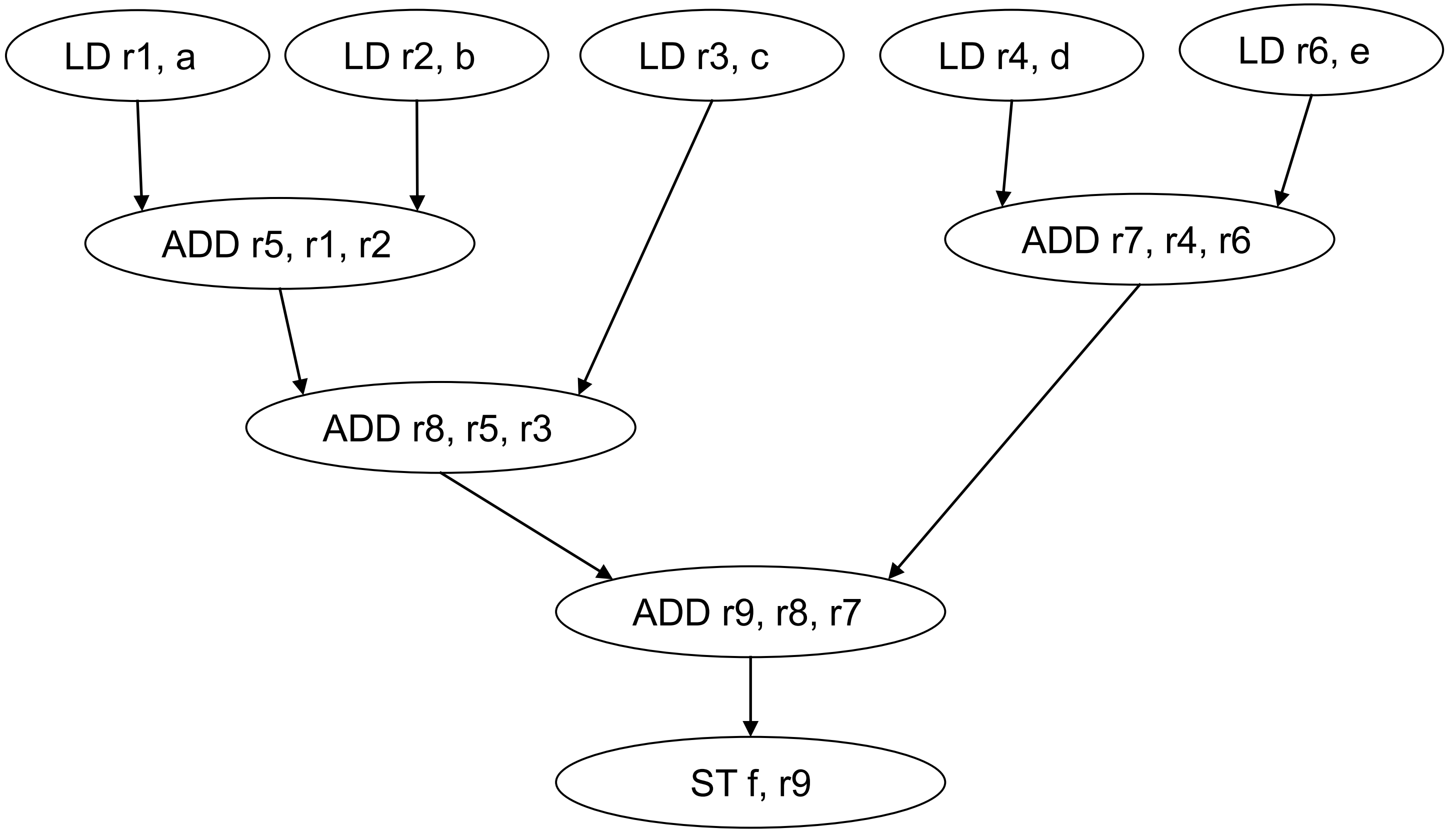
Expression tree



Assembly Code

```
LD r1, a
LD r2, b
ADD r5, r1, r2
LD r3, c
ADD r8, r5, r3
LD r4, d
LD r6, e
ADD r7, r4, r6
ADD r9, r8, r7
ST f, r9
```

DDG



List Scheduling

- One of the most used algorithms for scheduling the instructions in basic blocks
  - Local greedy heuristics
  - Instructions are prioritized - different priority functions can be used
  - Simple and highly effective

List Scheduling

- ***s***=1 // clock cycle
- ***Ready*** = nodes ready to be scheduled //and topologically ordered
- ***Active*** = ∅
- while ***Ready*** and ***Active*** are not empty
  - *foreach* node *n* in ***Active*** *and* ***S(n) + λ(n) ≤ s***
    - *Active* = *Active – n*
    - *foreach* m ∈ *succ*(n) and *isready*(m)
      - Ready = Ready ∪ m
    - *endforeach*
  - *endforeach*
  - *foreach* n ∈ **Ready** *by priority order and n can be assigned to a resource*
    - Ready = *Ready – n*
    - *Active* = *Active* ∪ *n*
    - *S(n)* = *s*
  - *endforeach*
  - *s* = *s* + 1;
- *endwhile*

λ(n): number of clock cycles of n
S(n): clock cycle (s) where n was scheduled

List Scheduling

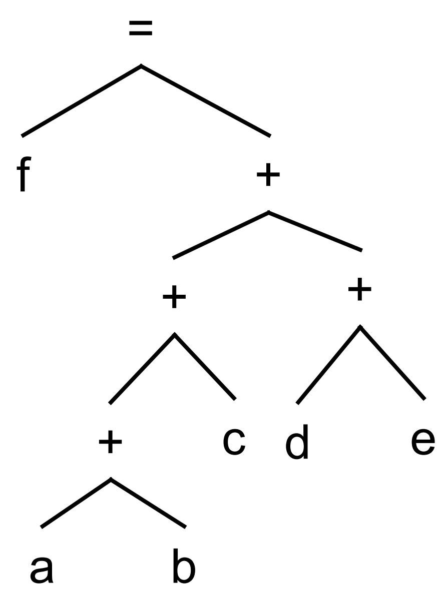
- Priorities
  - Longest latency path or critical path (height)
  - Mobility
- Tie-breaks
  - Last use of a value - decreases demand for register as moves it nearer def
  - Number of descendants - encourages scheduler to pursue multiple paths
  - Longer latency first - others can fit in shadow
  - Random

DDG for Example E1

Statement

f = (a + b) + c + (d + e)

Expression tree

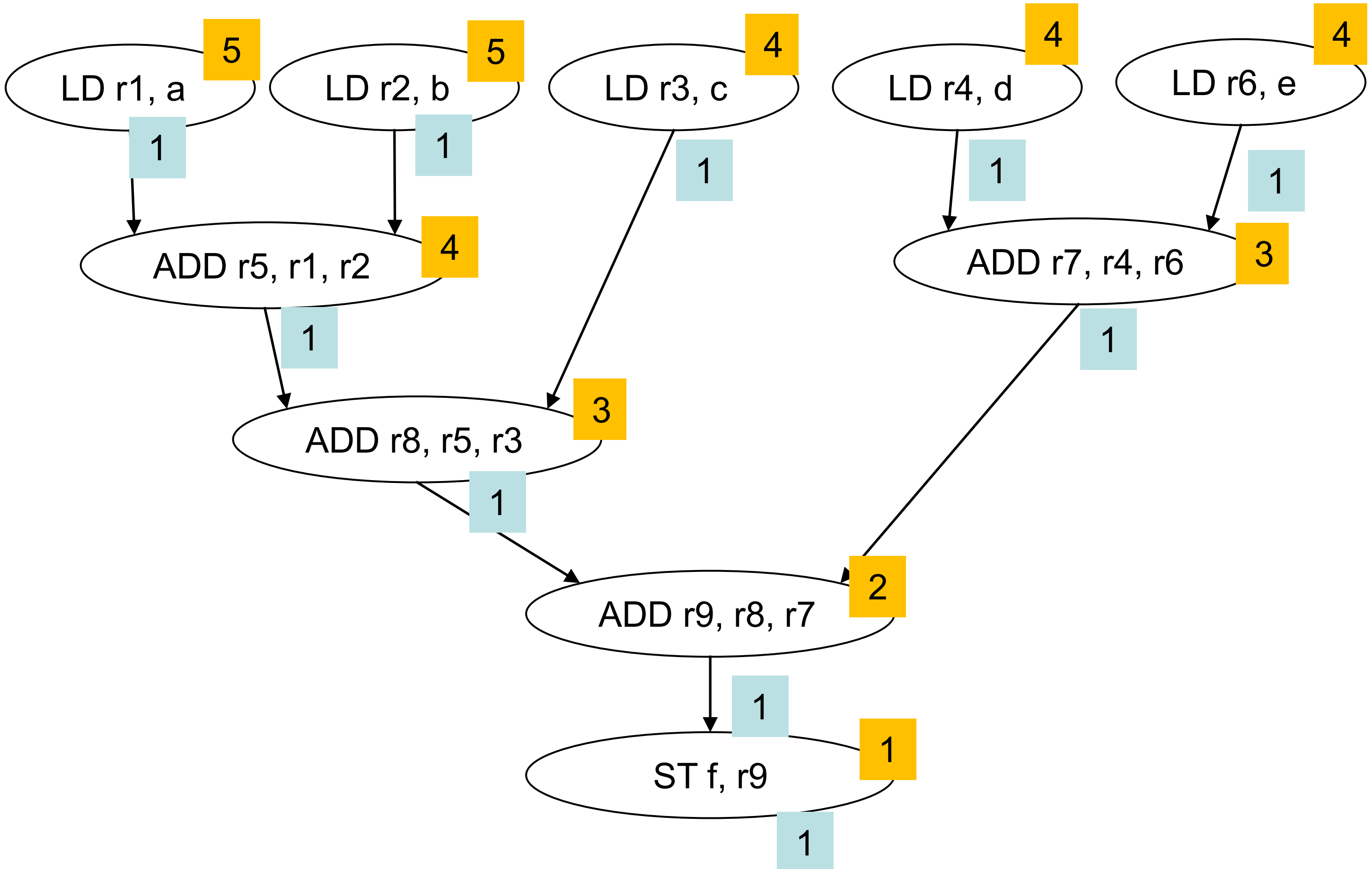


Assembly Code

```
LD r1, a
LD r2, b
ADD r5, r1, r2
LD r3, c
ADD r8, r5, r3
LD r4, d
LD r6, e
ADD r7, r4, r6
ADD r9, r8, r7
ST f, r9
```

DDG w/ clock cycles for each instruction and longest latencies

(1 clock cycle of latency for each instruction)



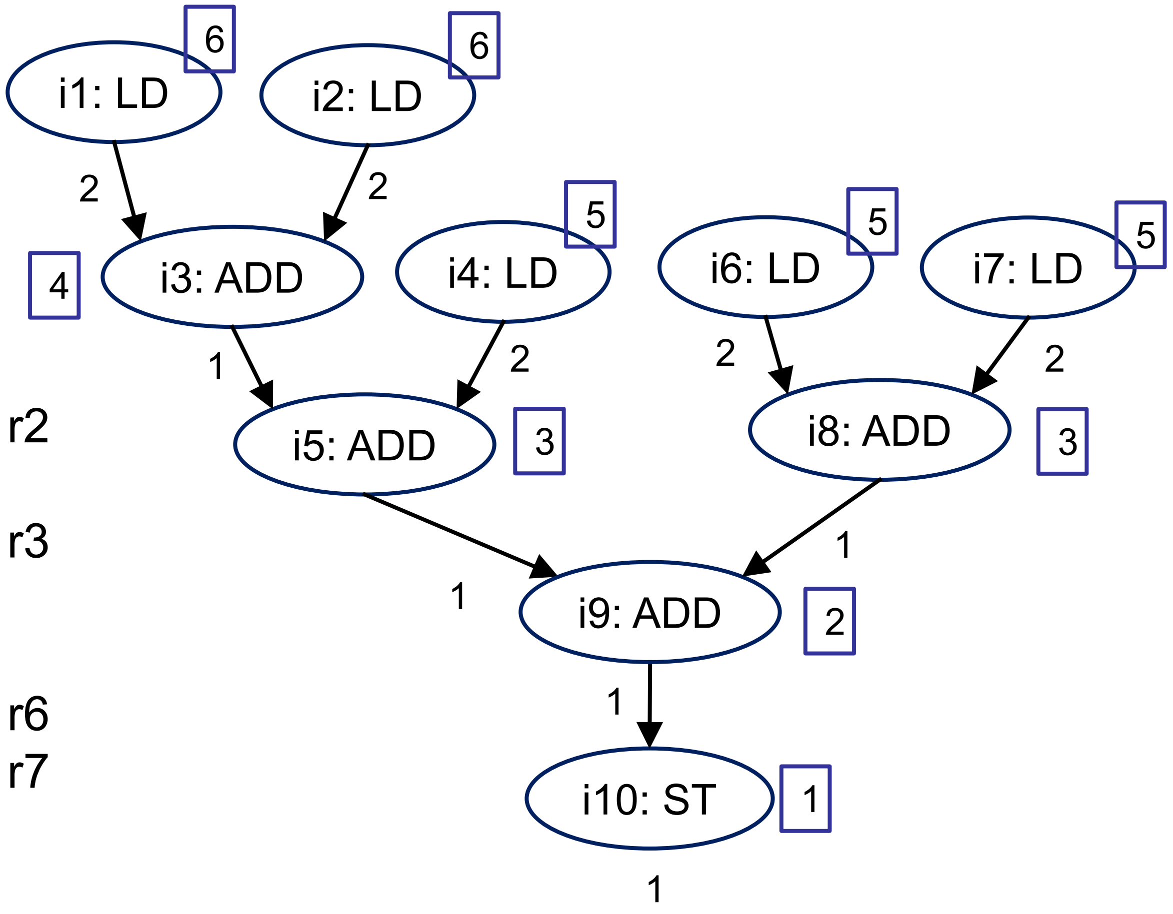
List Scheduling: Example E2

f = (a + b) + c + (d + e)

Assembly Code

i1: LD r1, a
i2: LD r2, b
i3: ADD r5, r1, r2
i4: LD r3, c
i5: ADD r8, r5, r3
i6: LD r4, d
i7: LD r6, e
i8: ADD r7, r4, r6
i9: ADD r9, r8, r7
i10: ST f, r9

Data-dependence graph (DDG)



Machine model:

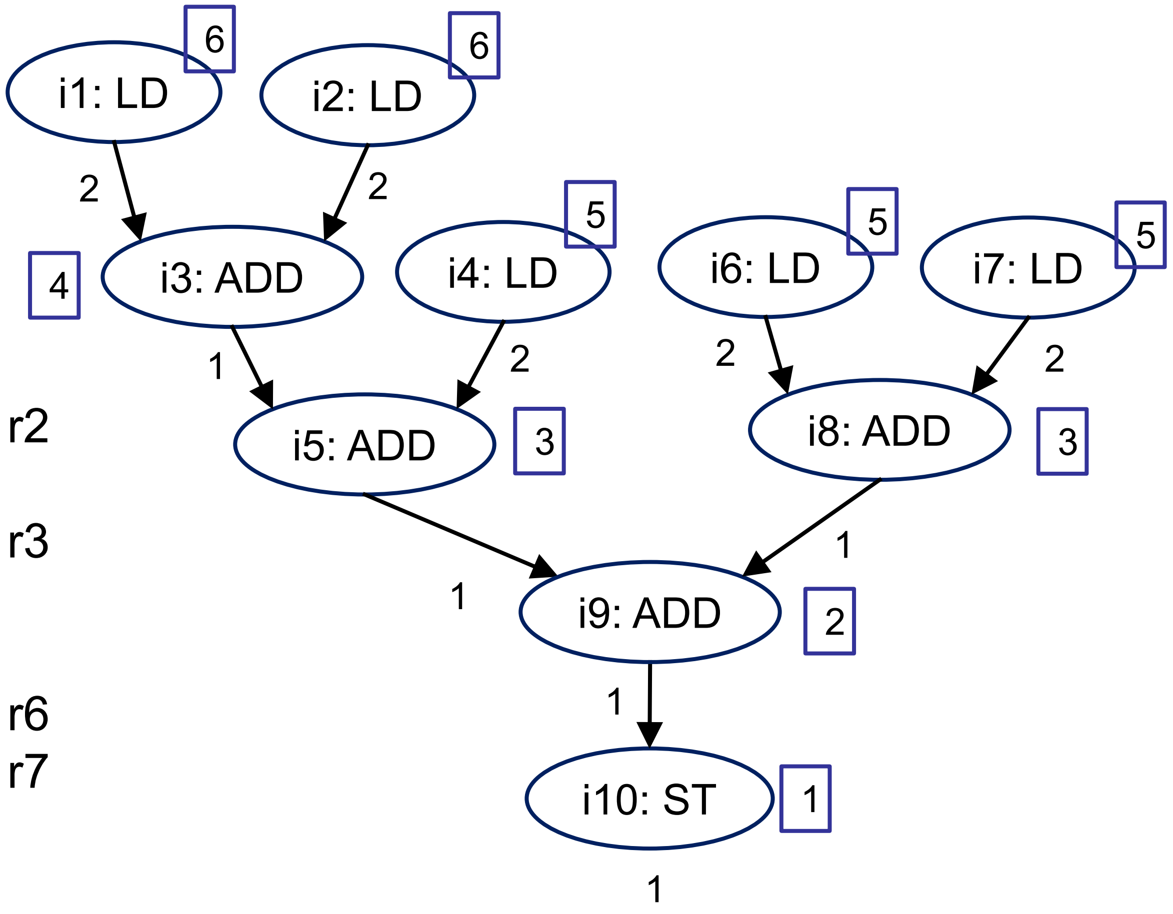
- Two ALU resources (for ADD, SUB, etc.)
- Two MEM resources (for LD and ST operations)
- All operations require 1 clock cycle, except LD, which requires 2 clock cycles
- An LD/ST can begin 1 clock cycle after an LD

List Scheduling: Example E2

Nodes ready to be scheduled and ordered by height

f = (a + b) + c + (d + e)

Data-dependence graph (DDG)



Assembly Code

i1: LD r1, a
i2: LD r2, b
i3: ADD r5, r1, r2
i4: LD r3, c
i5: ADD r8, r5, r3
i6: LD r4, d
i7: LD r6, e
i8: ADD r7, r4, r6
i9: ADD r9, r8, r7
i10: ST f, r9

Cycle: 1 Ready: 1, 2, 4, 6, 7
Active:

Cycle	MEM	MEM	ALU	ALU
1				
2				
3				
4				
5				
6				
7				

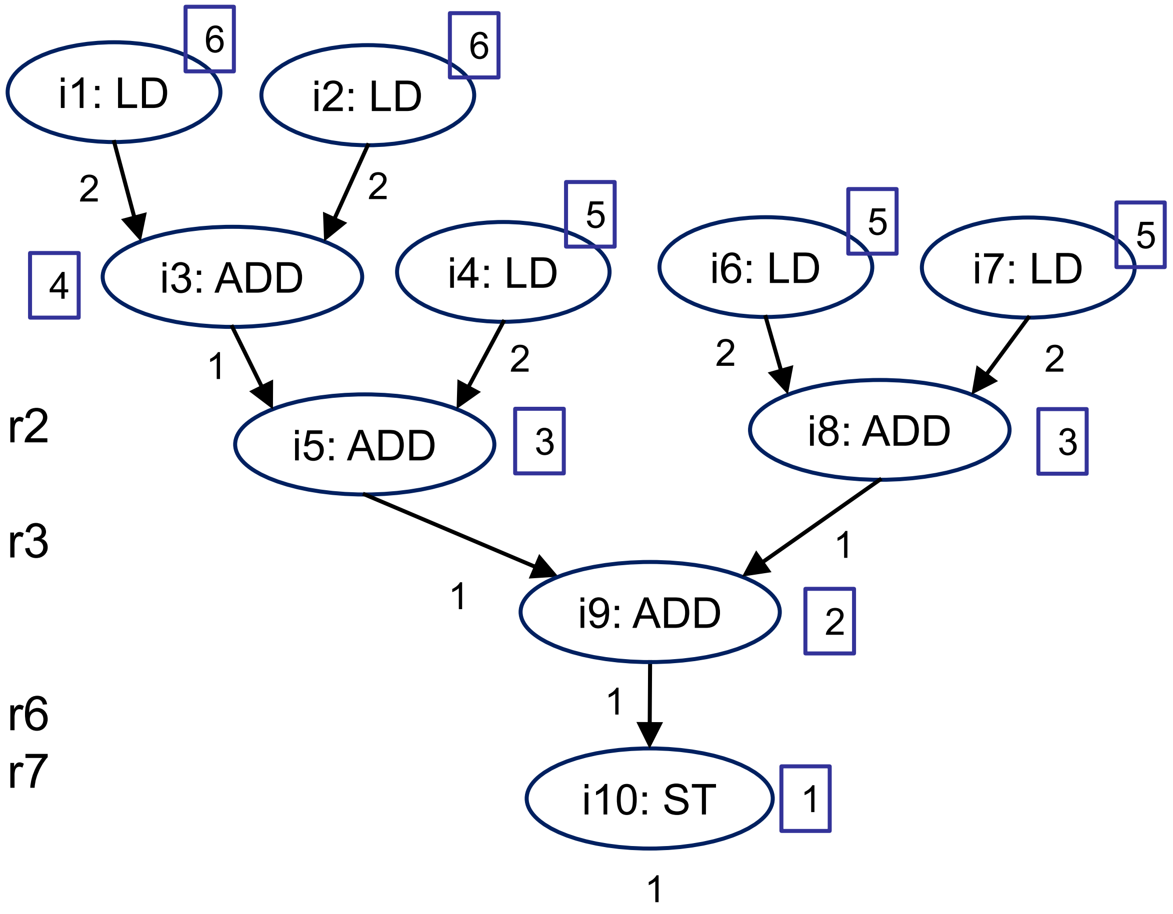
List Scheduling: Example E2

f = (a + b) + c + (d + e)

Assembly Code

i1: LD r1, a
i2: LD r2, b
i3: ADD r5, r1, r2
i4: LD r3, c
i5: ADD r8, r5, r3
i6: LD r4, d
i7: LD r6, e
i8: ADD r7, r4, r6
i9: ADD r9, r8, r7
i10: ST f, r9

Data-dependence graph (DDG)



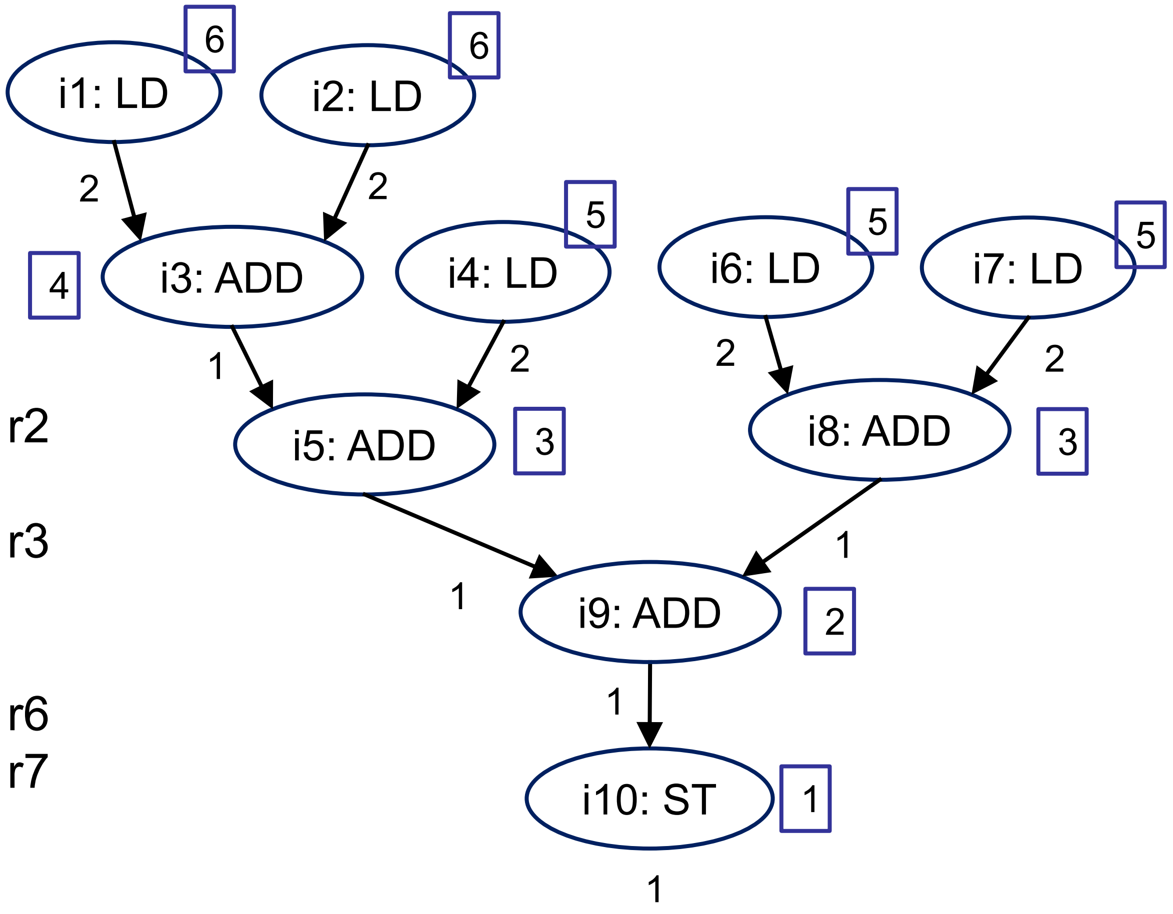
Cycle: 1 Ready: 4, 6, 7
Active: 1, 2

Cycle	MEM	MEM	ALU	ALU
1	1	2		
2				
3				
4				
5				
6				
7				

List Scheduling: Example E2

f = (a + b) + c + (d + e)

Data-dependence graph (DDG)



Assembly Code

i1: LD r1, a
i2: LD r2, b
i3: ADD r5, r1, r2
i4: LD r3, c
i5: ADD r8, r5, r3
i6: LD r4, d
i7: LD r6, e
i8: ADD r7, r4, r6
i9: ADD r9, r8, r7
i10: ST f, r9

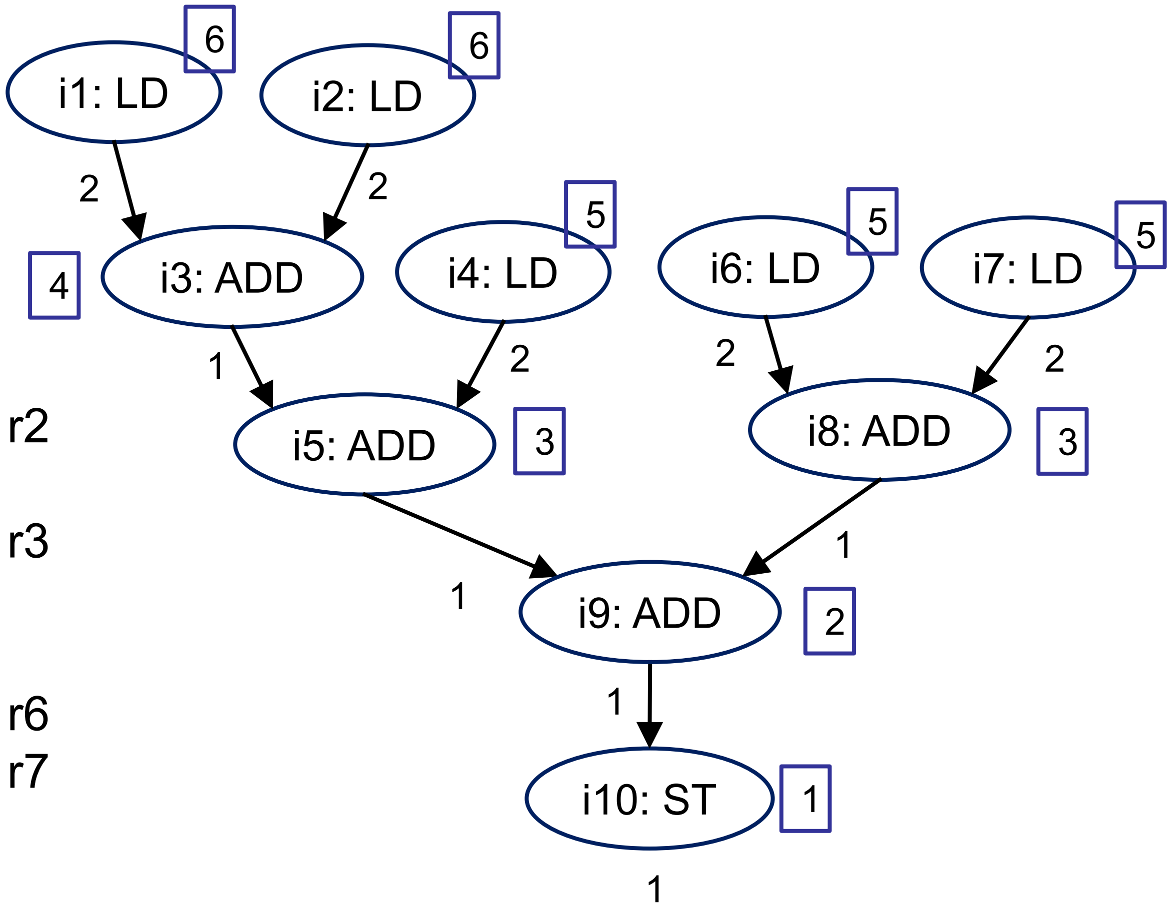
Cycle: 2 Ready: 4, 6, 7
Active: 1, 2

Cycle	MEM	MEM	ALU	ALU
1	1	2		
2				
3				
4				
5				
6				
7				

List Scheduling: Example E2

f = (a + b) + c + (d + e)

Data-dependence graph (DDG)



Assembly Code

i1: LD r1, a
i2: LD r2, b
i3: ADD r5, r1, r2
i4: LD r3, c
i5: ADD r8, r5, r3
i6: LD r4, d
i7: LD r6, e
i8: ADD r7, r4, r6
i9: ADD r9, r8, r7
i10: ST f, r9

Cycle: 2 Ready: 7
Active: 1, 2, 4, 6

Cycle	MEM	MEM	ALU	ALU
1	1	2		
2	4	6		
3				
4				
5				
6				
7				

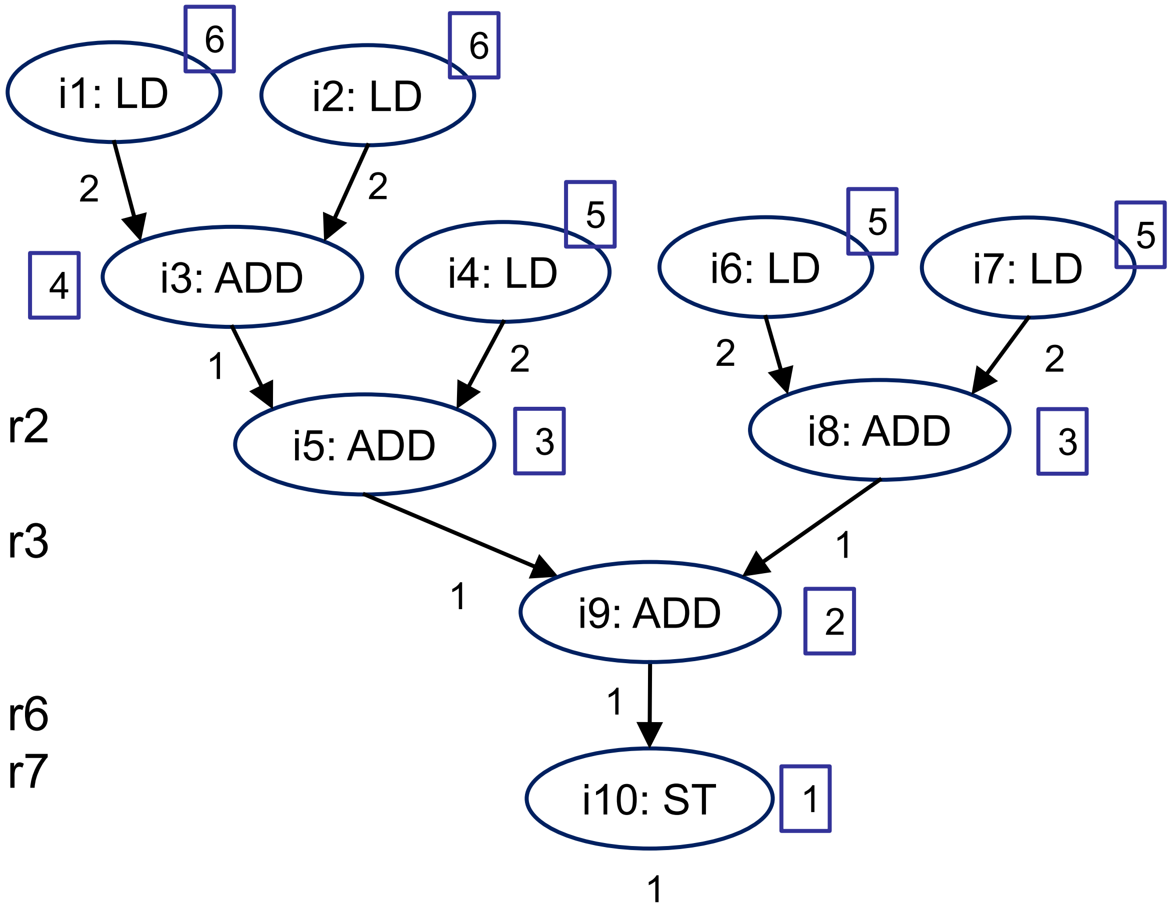
List Scheduling: Example E2

f = (a + b) + c + (d + e)

Assembly Code

i1: LD r1, a
i2: LD r2, b
i3: ADD r5, r1, r2
i4: LD r3, c
i5: ADD r8, r5, r3
i6: LD r4, d
i7: LD r6, e
i8: ADD r7, r4, r6
i9: ADD r9, r8, r7
i10: ST f, r9

Data-dependence graph (DDG)



Cycle: 3 Ready: 7, 3
Active: 4, 6

Cycle	MEM	MEM	ALU	ALU
1	1	2		
2	4	6		
3				
4				
5				
6				
7				

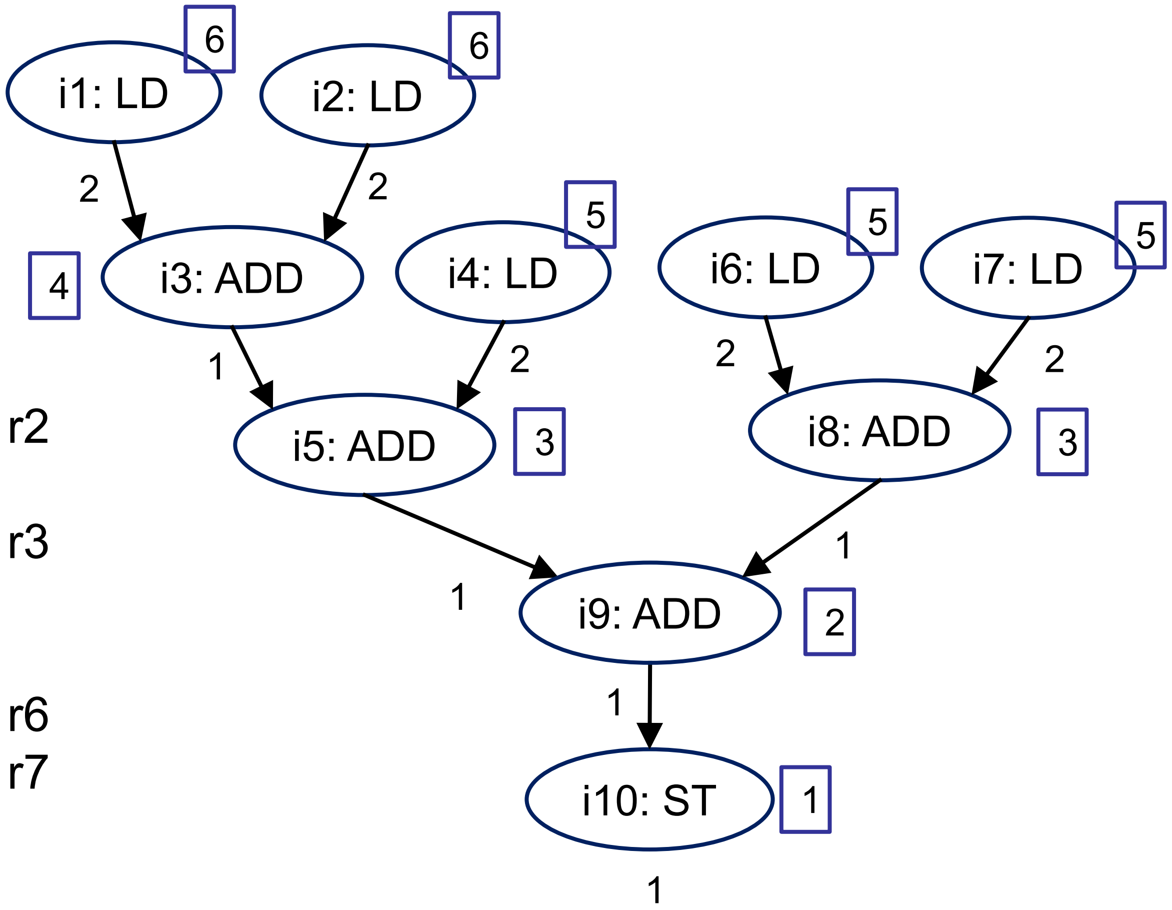
List Scheduling: Example E2

f = (a + b) + c + (d + e)

Assembly
Code

i1: LD r1, a
i2: LD r2, b
i3: ADD r5, r1, r2
i4: LD r3, c
i5: ADD r8, r5, r3
i6: LD r4, d
i7: LD r6, e
i8: ADD r7, r4, r6
i9: ADD r9, r8, r7
i10: ST f, r9

Data-dependence graph (DDG)



Cycle: 3 Ready:
Active: 4, 6, 7, 3

Cycle	MEM	MEM	ALU	ALU
1	1	2		
2	4	6		
3	7		3	
4				
5				
6				
7				

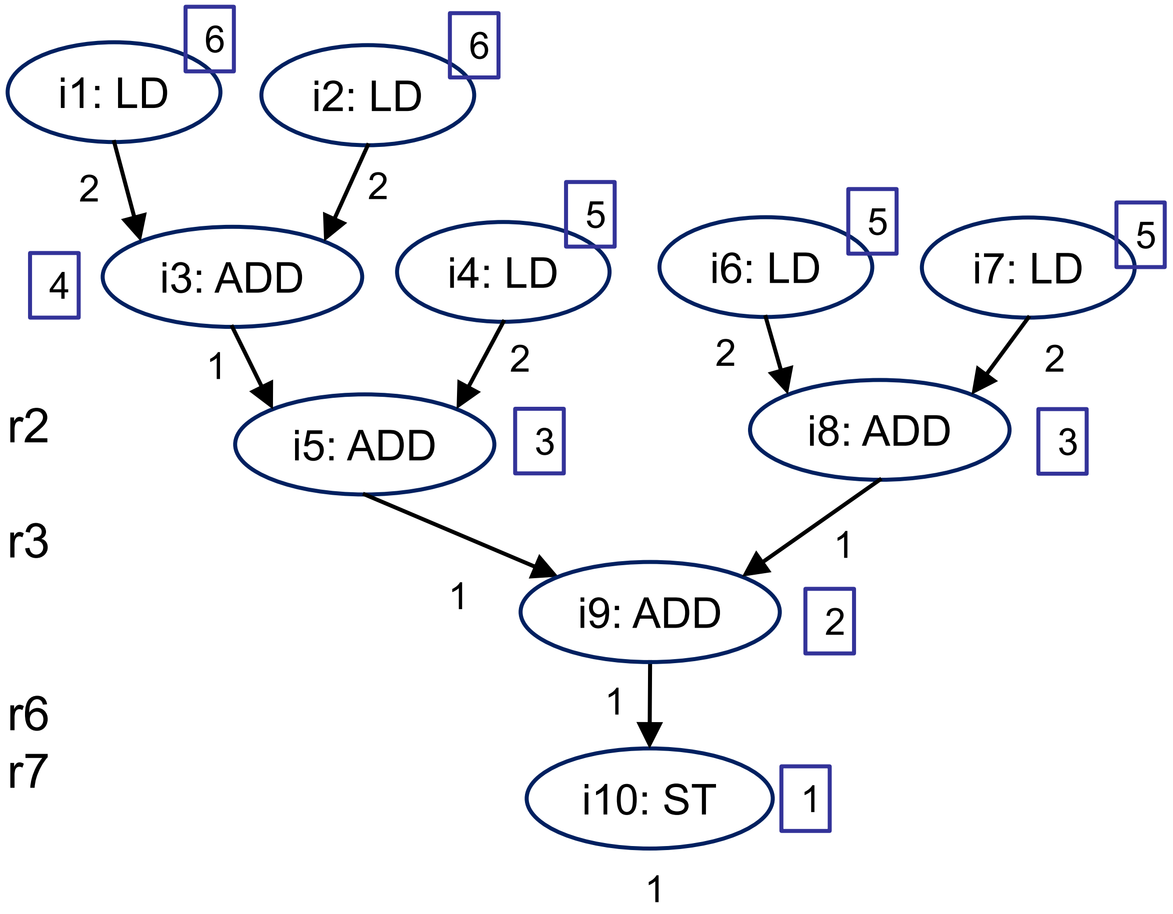
List Scheduling: Example E2

f = (a + b) + c + (d + e)

Assembly Code

i1: LD r1, a
i2: LD r2, b
i3: ADD r5, r1, r2
i4: LD r3, c
i5: ADD r8, r5, r3
i6: LD r4, d
i7: LD r6, e
i8: ADD r7, r4, r6
i9: ADD r9, r8, r7
i10: ST f, r9

Data-dependence graph (DDG)



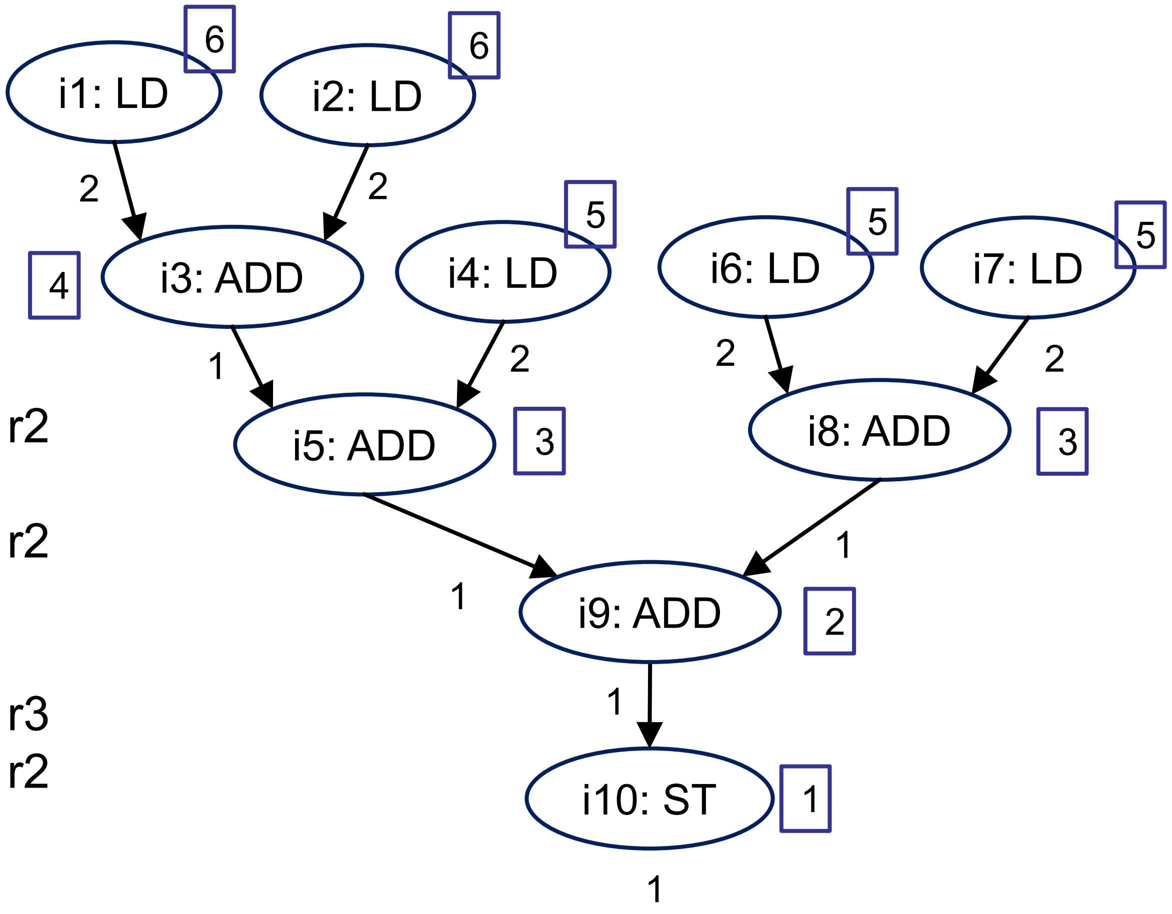
Cycle: 4 Ready: 5
Active: 7

Cycle	MEM	MEM	ALU	ALU
1	1	2		
2	4	6		
3	7		3	
4				
5				
6				
7				

List Scheduling: Example E2

f = (a + b) + c + (d + e)

Data-dependence graph (DDG)



Assembly Code

i1: LD r1, a
i2: LD r2, b
i3: ADD r1, r1, r2
i4: LD r2, c
i5: ADD r1, r1, r2
i6: LD r2, d
i7: LD r3, e
i8: ADD r2, r2, r3
i9: ADD r1, r1, r2
i10: ST f, r1

Cycle: 4 Ready:
Active: 7, 5

Cycle	MEM	MEM	ALU	ALU
1	1	2		
2	4	6		
3	7		3	
4			5	
5				
6				
7				

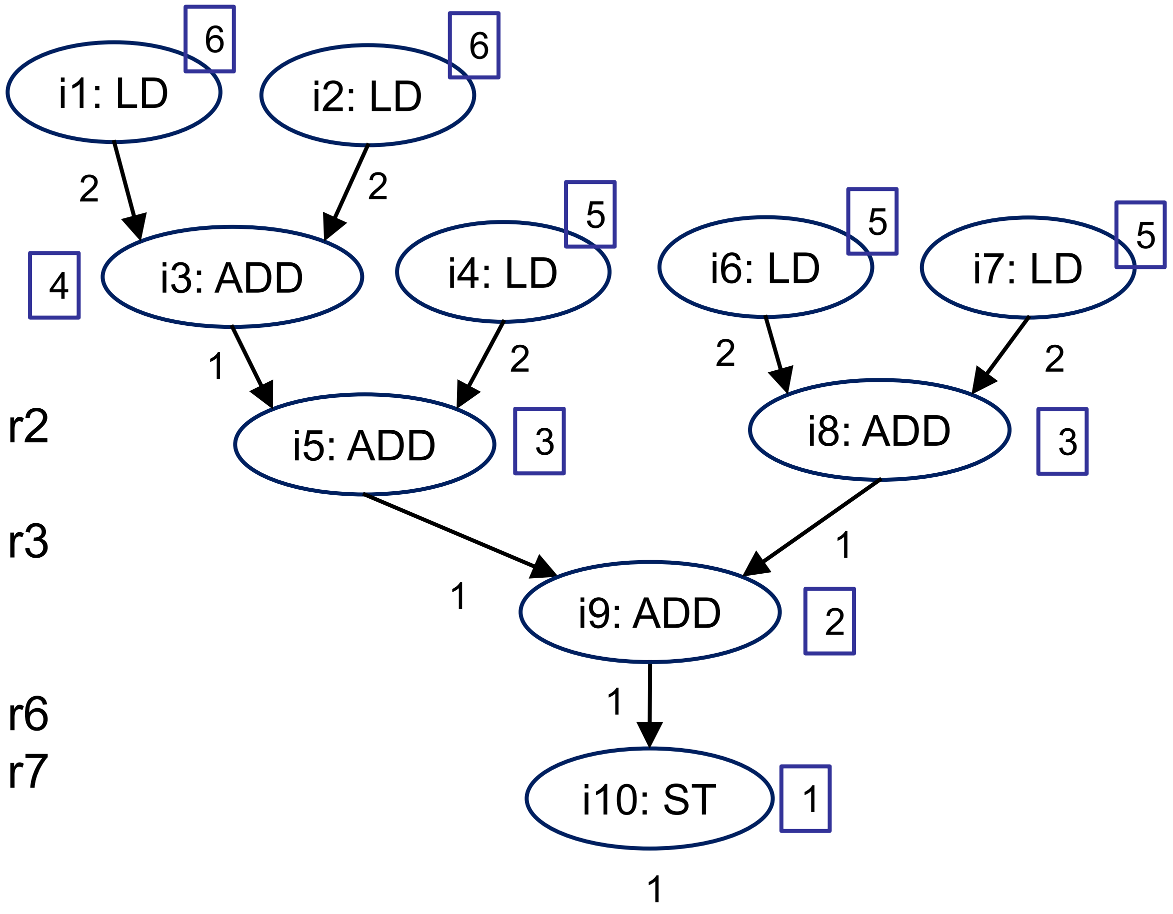
List Scheduling: Example E2

f = (a + b) + c + (d + e)

Assembly Code

i1: LD r1, a
i2: LD r2, b
i3: ADD r5, r1, r2
i4: LD r3, c
i5: ADD r8, r5, r3
i6: LD r4, d
i7: LD r6, e
i8: ADD r7, r4, r6
i9: ADD r9, r8, r7
i10: ST f, r9

Data-dependence graph (DDG)



Cycle: 5 Ready: 8
Active:

Cycle	MEM	MEM	ALU	ALU
1	1	2		
2	4	6		
3	7		3	
4			5	
5				
6				
7				

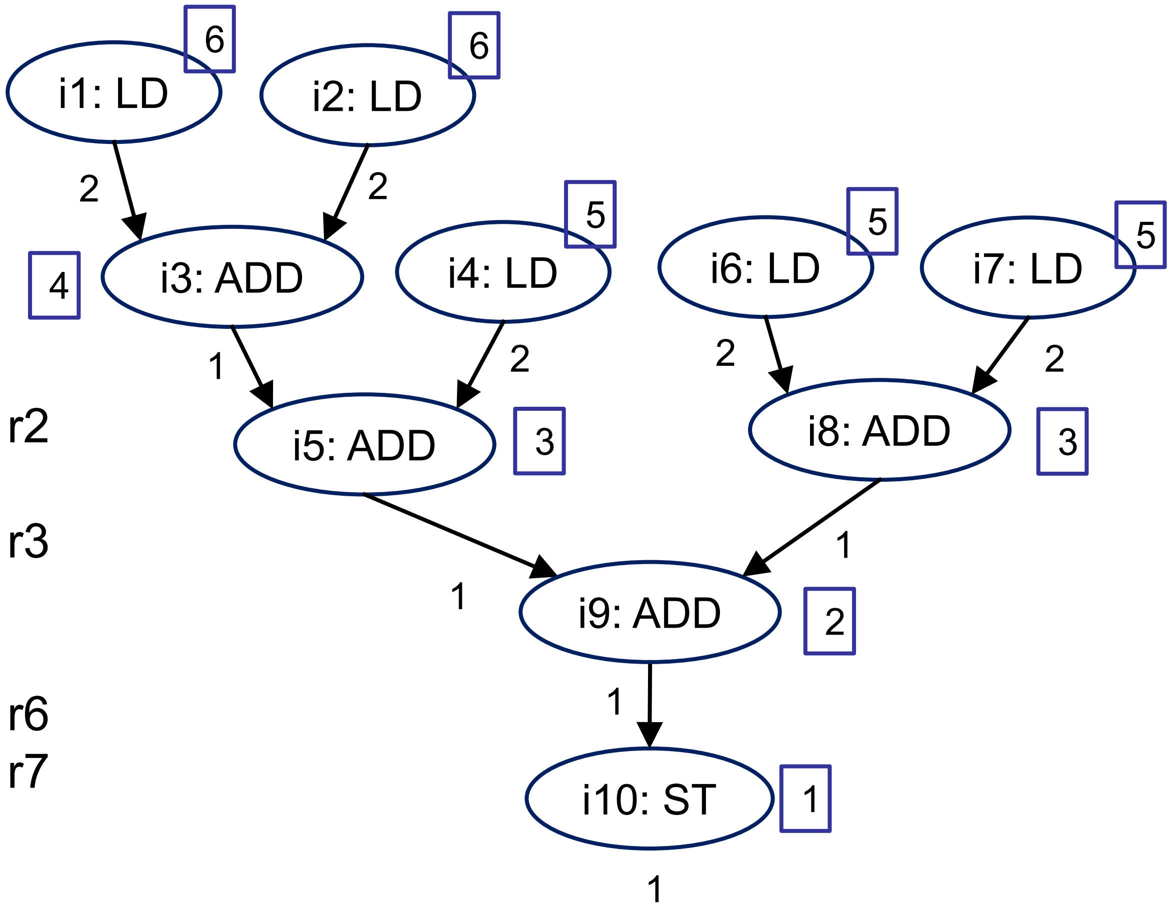
List Scheduling: Example E2

f = (a + b) + c + (d + e)

Assembly
Code

i1: LD r1, a
i2: LD r2, b
i3: ADD r5, r1, r2
i4: LD r3, c
i5: ADD r8, r5, r3
i6: LD r4, d
i7: LD r6, e
i8: ADD r7, r4, r6
i9: ADD r9, r8, r7
i10: ST f, r9

Data-dependence graph (DDG)



Cycle: 5   Ready:
Active: 8

Cycle	MEM	MEM	ALU	ALU
1	1	2		
2	4	6		
3	7		3	
4			5	
5			8	
6				
7				

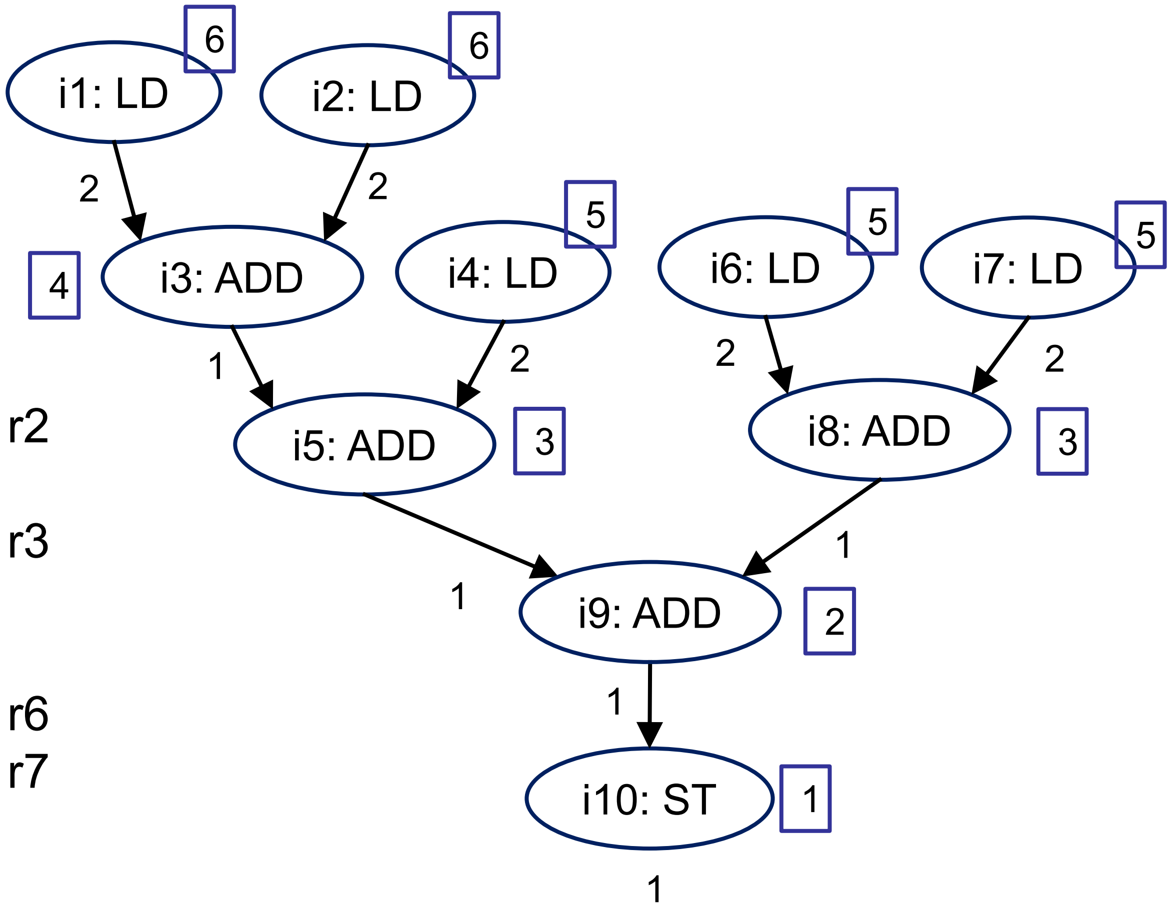
List Scheduling: Example E2

f = (a + b) + c + (d + e)

Assembly Code

i1: LD r1, a
i2: LD r2, b
i3: ADD r5, r1, r2
i4: LD r3, c
i5: ADD r8, r5, r3
i6: LD r4, d
i7: LD r6, e
i8: ADD r7, r4, r6
i9: ADD r9, r8, r7
i10: ST f, r9

Data-dependence graph (DDG)



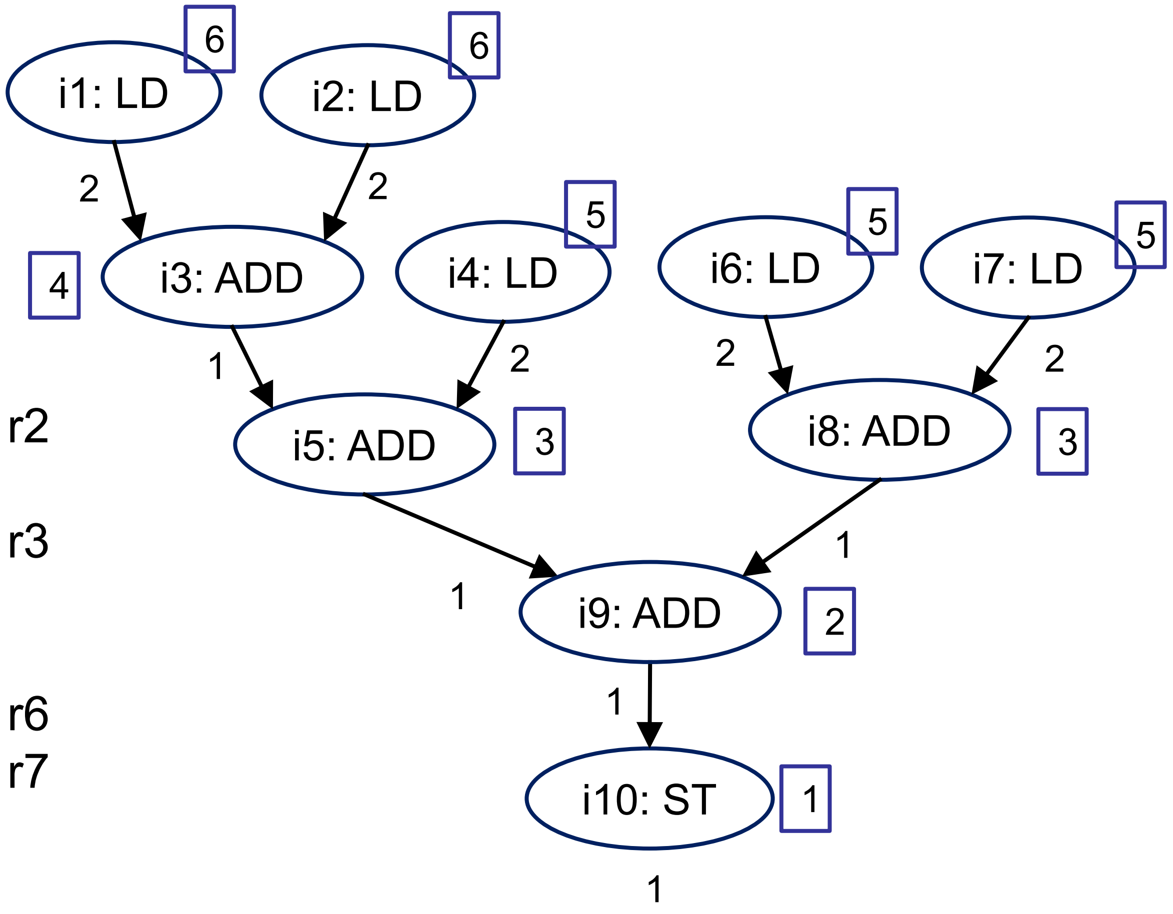
Cycle: 6 Ready: 9
Active:

Cycle	MEM	MEM	ALU	ALU
1	1	2		
2	4	6		
3	7		3	
4			5	
5			8	
6				
7				

List Scheduling: Example E2

f = (a + b) + c + (d + e)

Data-dependence graph (DDG)



Assembly Code

i1: LD r1, a
i2: LD r2, b
i3: ADD r5, r1, r2
i4: LD r3, c
i5: ADD r8, r5, r3
i6: LD r4, d
i7: LD r6, e
i8: ADD r7, r4, r6
i9: ADD r9, r8, r7
i10: ST f, r9

Cycle: 6 Ready:
Active: 9

Cycle	MEM	MEM	ALU	ALU
1	1	2		
2	4	6		
3	7		3	
4			5	
5			8	
6			9	
7				

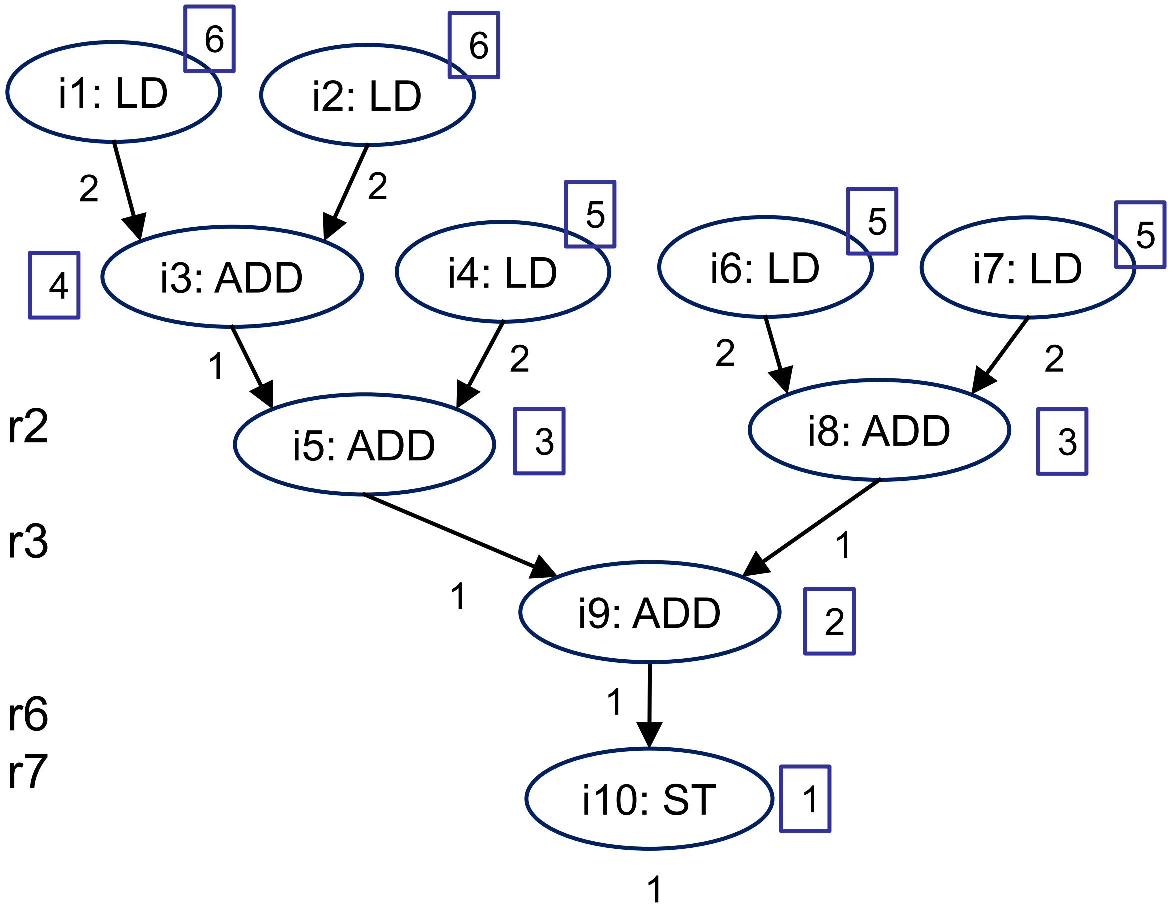
List Scheduling: Example E2

f = (a + b) + c + (d + e)

Assembly Code

i1: LD r1, a
i2: LD r2, b
i3: ADD r5, r1, r2
i4: LD r3, c
i5: ADD r8, r5, r3
i6: LD r4, d
i7: LD r6, e
i8: ADD r7, r4, r6
i9: ADD r9, r8, r7
i10: ST f, r9

Data-dependence graph (DDG)



Cycle: 7 Ready: 10
Active:

Cycle	MEM	MEM	ALU	ALU
1	1	2		
2	4	6		
3	7		3	
4			5	
5			8	
6			9	
7				

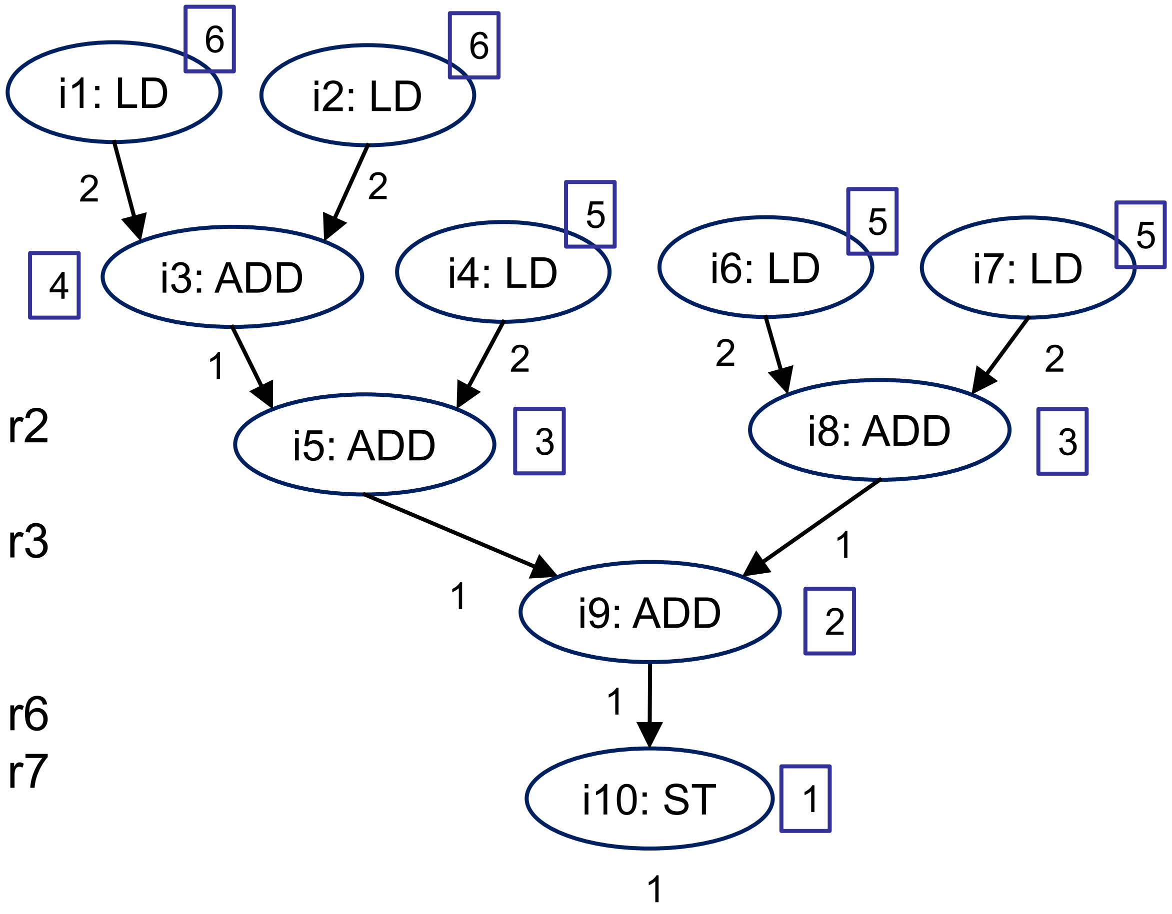
List Scheduling: Example E2

f = (a + b) + c + (d + e)

Assembly
Code

i1: LD r1, a
i2: LD r2, b
i3: ADD r5, r1, r2
i4: LD r3, c
i5: ADD r8, r5, r3
i6: LD r4, d
i7: LD r6, e
i8: ADD r7, r4, r6
i9: ADD r9, r8, r7
i10: ST f, r9

Data-dependence graph (DDG)



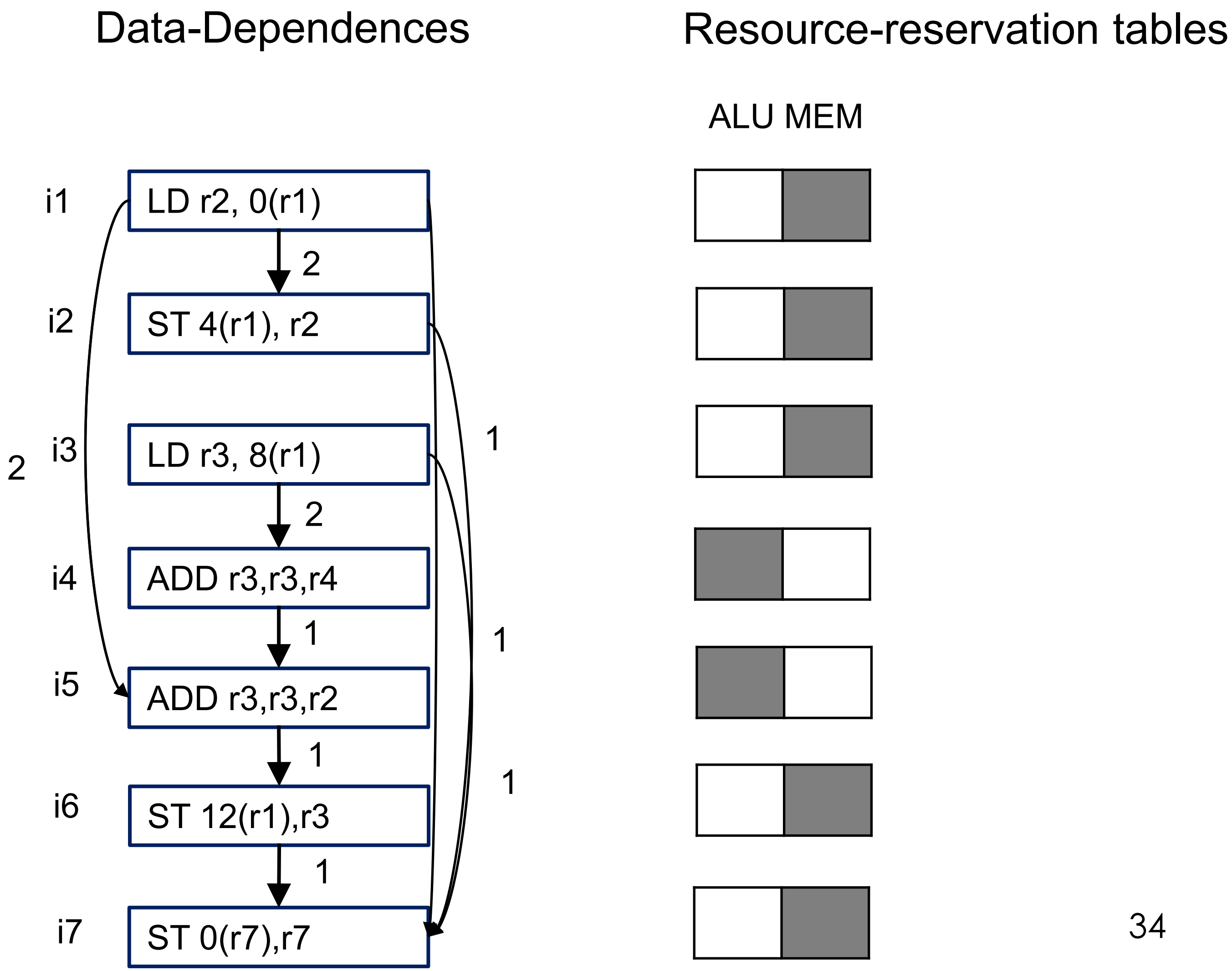
Cycle: 7 Ready:
Active: 10

Cycle	MEM	MEM	ALU	ALU
1	1	2		
2	4	6		
3	7		3	
4			5	
5			8	
6			9	
7	10			

Example E3

Machine model:
- One ALU resource (for ADD, SUB, etc.)
- One MEM resource (for LD and ST operations)
- All operations require 1 clock cycle, except LD, which requires 2 clock cycles
- An LD/ST can begin 1 clock cycle after an LD

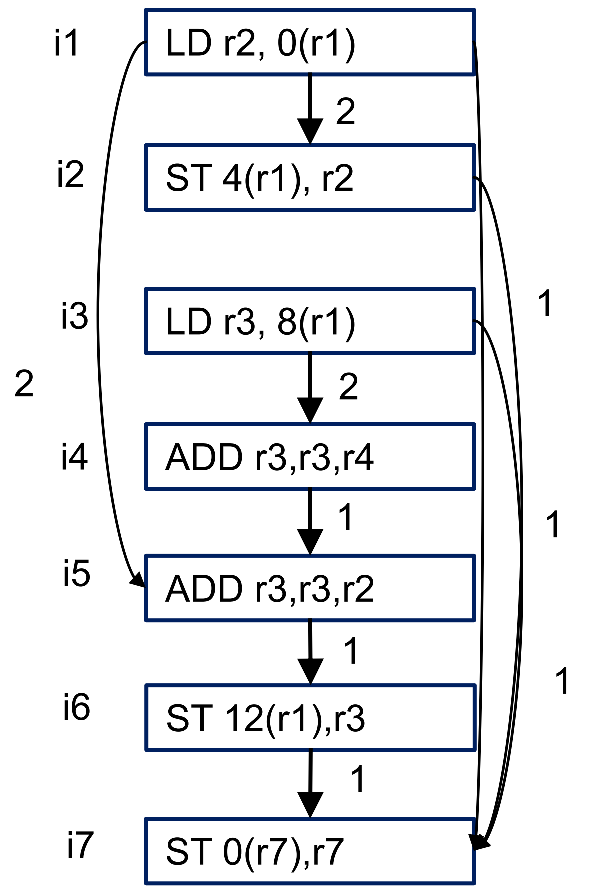
Critical path: 6 clock cycles



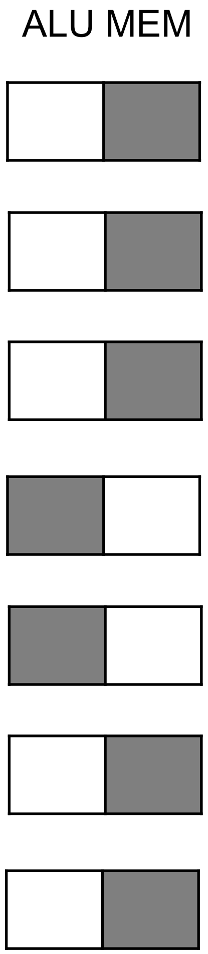
Example E3

- List scheduling result

Data-Dependences



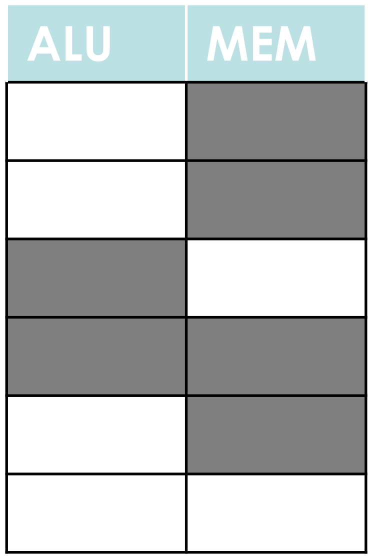
Resource-reservation tables



Schedule

ALU	MEM
	LD r3, 8(r1)
	LD r2, 0(r1)
ADD r3, r3, r4	
ADD r3, r3, r2	ST 4(r1), r2
	ST 12(r1), r3
	ST 0(r7), r7

Resource-reservation table



Critical path: 6 clock cycles

Exercise 2

- Consider the prevous code example and the respective DDG and Schedule it using list-scheduling assuming:
  - a) the machine has only one ALU resource and two MEM resources
  - b) the machine has two ALU resources and one MEM resources
  - c) machine has two ALU resources and two MEM resources

Exercise 3

- Assuming the previous machine model and the input three address code below:
  - a) draw the data dependence graph
  - b) identify all the critical paths in your graph from part (a)
  - c) Assuming unlimited MEM resources, what are all the possible schedules for the 7 instructions?

Three-address code

```
LD r1, a
ST b, r1
LD r2, c
ST c, r1
LD r1, d
ST d, r2
ST a, r1
```

Location of Scheduling

- Phase ordering of register allocation and scheduling
  - Scheduling before register allocation
  - Scheduling after register allocation
- Think about the advantages and disadvantages of each ordering above
- Why not to solve both (scheduling and register allocation) at the same time?
- And what about instruction selection? The selection of instructions impacts scheduling and register allocation!

Parallelism can be explored across basic blocks

- Using code regions consisting of sets of basic blocks
  - e.g., using the extended basic block
- Extended Basic Block
  - Maximum set of basic blocks where the entry is a single basic block and each other basic block has only one predecessor

Parallelism can be explored across basic blocks (cont.)

- Requires global code scheduling
- Control dependences

If (c) s1; else s2;	s1 and s2 are control dependent on c	
while (c) s;	the body s is control dependent on c	

if (a > t)
b=a*a;
d = a+c;
	the statements b=a*a and d=a+c have no data dependence	d=a+c can be scheduled any time
		Assuming that * does not cause any side-effect, a*a can be speculatively executed

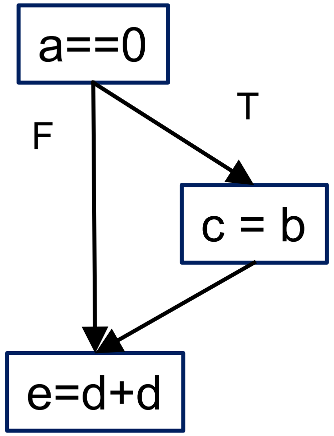
Global code scheduling

- Considers code motion (upward, downward)
- Considers speculative execution of instructions without side-effects

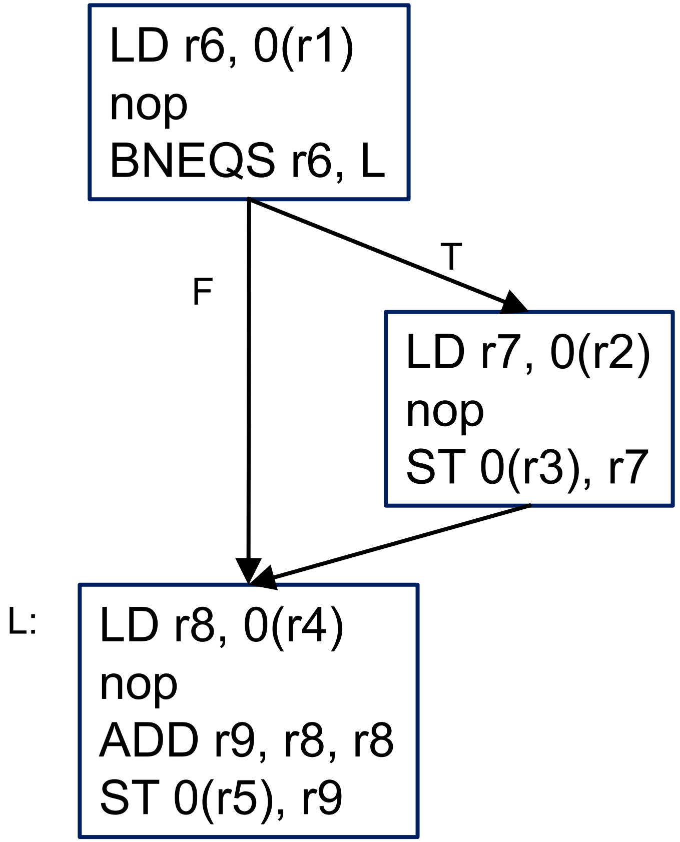
Global Scheduling

- Suppose a machine that can execute any 2 operations in a single clock. Every operation executes with a delay of 1 clock cycle, except the load operations that have a latency of 2 clock cycles.
- For simplicity we assume that all memory accesses in the example are valid and will hit in the cache.

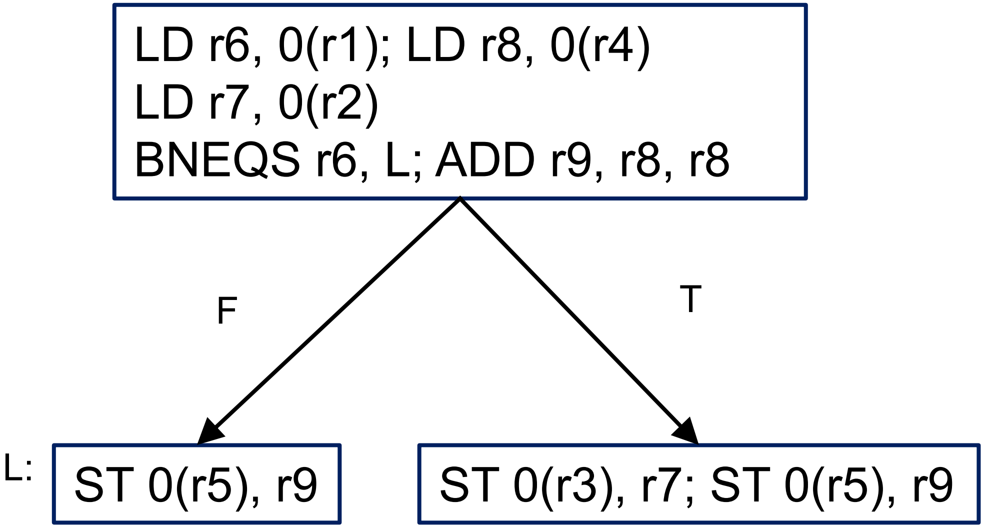
Input code in a CFG



CFG with machine code locally scheduled



CFG with machine code globally scheduled



Software Pipelining

- Important compiler optimization that overlaps (fully of partially) successive iterations (i.e., the subsequent iteration may start before the previous one is finished)
- This example shows, at source code level, the overlapping of 2 consecutive iterations

```
for(i=0;i<1000; i++)
   C[i]=A[i]+B[i];
```

Latency* = 1000*3 = 3000 cycles

```
t1 = A[0]; t2 = B[0];                  // prologue
for(i=1;i<1000; i++) {
   t3 = t1+t2; t1 = A[i]; t2 = B[i];
   C[i-1] = t3;                         // kernel
}
t3 =  t1+t2;                            // epilogue
C[999] = t3;                            // epilogue
```

Latency** = 1+999*2+2 = 2001 cycles

*,** Rough estimations based on the high-level statements and considering 1 clock cycle per operation
** also considering that operations at the same line are executed in parallel

Software Pipelining

- Important compiler optimization that overlaps (fully of partially) sucessive iterations (i.e., the subsequente iteration may start before the previous one is finished)
- This example shows, at source code level, the overlapping of 3 consecutive iterations

```
for(i=0;i<1000; i++)
   C[i]=A[i]+B[i];
```

Latency* = 1000*3 = 3000 cycles

```
t1 = A[0]; t2 = B[0];                    // prologue
t3 = t1 + t2; t1 = A[1]; t2 = B[1];
for(i=2;i<1000; i++) {
   C[i-2] = t3; t3 = t1+t2; t1 = A[i]; t2 = B[i]; // all in parallel
}
C[998] = t3; t3 =  t1+t2;                // epilogue
C[999] = t3;                             // epilogue
```

Latency** = 2+998*1+2 = 1002 cycles

*,** Rough estimations based on the high-level statements and considering 1 clock cycle per operation
** also considering that operations at the same line are executed in parallel

Increasing Opportunities for Instruction-Level Parallelism

- Some compiler optimizations increase the opportunities for instruction-level parallelism by exposing more instructions to be scheduled in a region
- Loop unrolling (partial or full) is one of them
  - E.g., loop unrolling by 2 duplicates the loop body and may expose more parallelism

```
for(i=0;i<1000; i++)
   C[i]=A[i]+B[i];
```

```
for(i=0;i<1000; i+=2) {
   C[i]=A[i]+B[i];
   C[i+1]=A[i+1]+B[i+1];
}
```

Pipelining stages

- The machine models used in the previous slides are simplistic
- In a typical microprocessor there are stages and instructions can be executed in pipelining
- E.g., in a typical 5-stage microprocessor architecture we have the following stages: IF, ID, EX, MEM, WB
- Scheduling needs
  - to consider the execution pipelining and
  - whenever possible avoiding to wait for the end of the execution of a previous instruction to start a new one
- When considering branch-delay slots, the scheduler also needs
  - to consider the speculative execution of instructions after the conditional branches and/or
  - the inclusions of *nop* instructions

Further Reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. **Compilers: Principles, Techniques, and Tools** (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Steven S. Muchnick. 1998. **Advanced Compiler Design and Implementation**. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- M. Lam. 1988. **Software pipelining: an effective scheduling technique for VLIW machines**. In Proceedings of the ACM SIGPLAN 1988 conference on Programming language design and implementation (PLDI '88), R. L. Wexelblat (Ed.). ACM, New York, NY, USA, 318-328. DOI=http://dx.doi.org/10.1145/53990.54022