CS406: Compilers

Spring 2022

Week 10: Register allocation, Instruction Scheduling

Slides Acknowledgements: Milind Kulkarni

Register Allocation

• Simple code generation (in CSE example): use a register for each temporary, load from a variable on each read, store to a variable at each write

•What are the problems?

- •Real machines have a limited number of registers – one register per temporary may be too many
- • Loading from and storing to variables on each use may produce a lot of redundant loads and stores

Register Allocation

- Goal: allocate temporaries and variables to registers to:
  - Use only as many registers as machine supports
  - Minimize loading and storing variables to memory (keep variables in registers when possible)
  - Minimize putting temporaries on stack (“spilling”)

Global vs. Local

- Same distinction as global vs. local CSE
  - Local register allocation is for a single basic block
  - Global register allocation is for an entire function

Does inter-procedural register allocation make sense? Why? Why not?

Hint: think about caller-save, callee-save registers

When we handle function calls, registers are pushed/popped from stack

Top-down register allocation

- For each basic block
  - Find the number of references of each variable
  - Assign registers to variables with the most references
- Details
  - Keep some registers free for operations on unassigned variables and spilling
  - Store *dirty* registers at the end of BB (i.e., registers which have variables assigned to them)
    - Do not need to do this for temporaries (why?)

Bottom-up register allocation

- Smarter approach:
  - Free registers once the data in them isn't used anymore
- Requires calculating *liveness*
  - A variable is live if it has a value that *may* be used in the future
- Easy to calculate if you have a single basic block:
  - Start at end of block, all local variables marked dead
    - If you have multiple basic blocks, all local variables defined in the block should be *live* (they may be used in the future)
  - When a variable is used, mark as live, record use
  - When a variable is defined, record def, variable dead above this
  - Creates chains linking uses of variables to where they were defined
- We will discuss how to calculate this across BBs later

Bottom-up register allocation

```
For each tuple op A B C in a BB, do
   Rx = ensure(A)
   Ry = ensure(B)
   if A dead after this tuple, free(Rx)
   if B dead after this tuple, free(Ry)
   Rz = allocate(C) //could use Rx or Ry
   generate code for op
   mark Rz dirty

At end of BB, for each dirty register
   generate code to store register into appropriate variable
```

- We will present this as if A, B, C are variables in memory. Can be modified to assume that A, B and C are in virtual registers, instead

Bottom-up register allocation

```
ensure(opr)
    if opr is already in register r
        return r
    else
        r = allocate(opr)
        generate load from opr into r
        return r
```

```
free(r)
    if r is marked dirty and variable is live
        generate store
    mark r as free
```

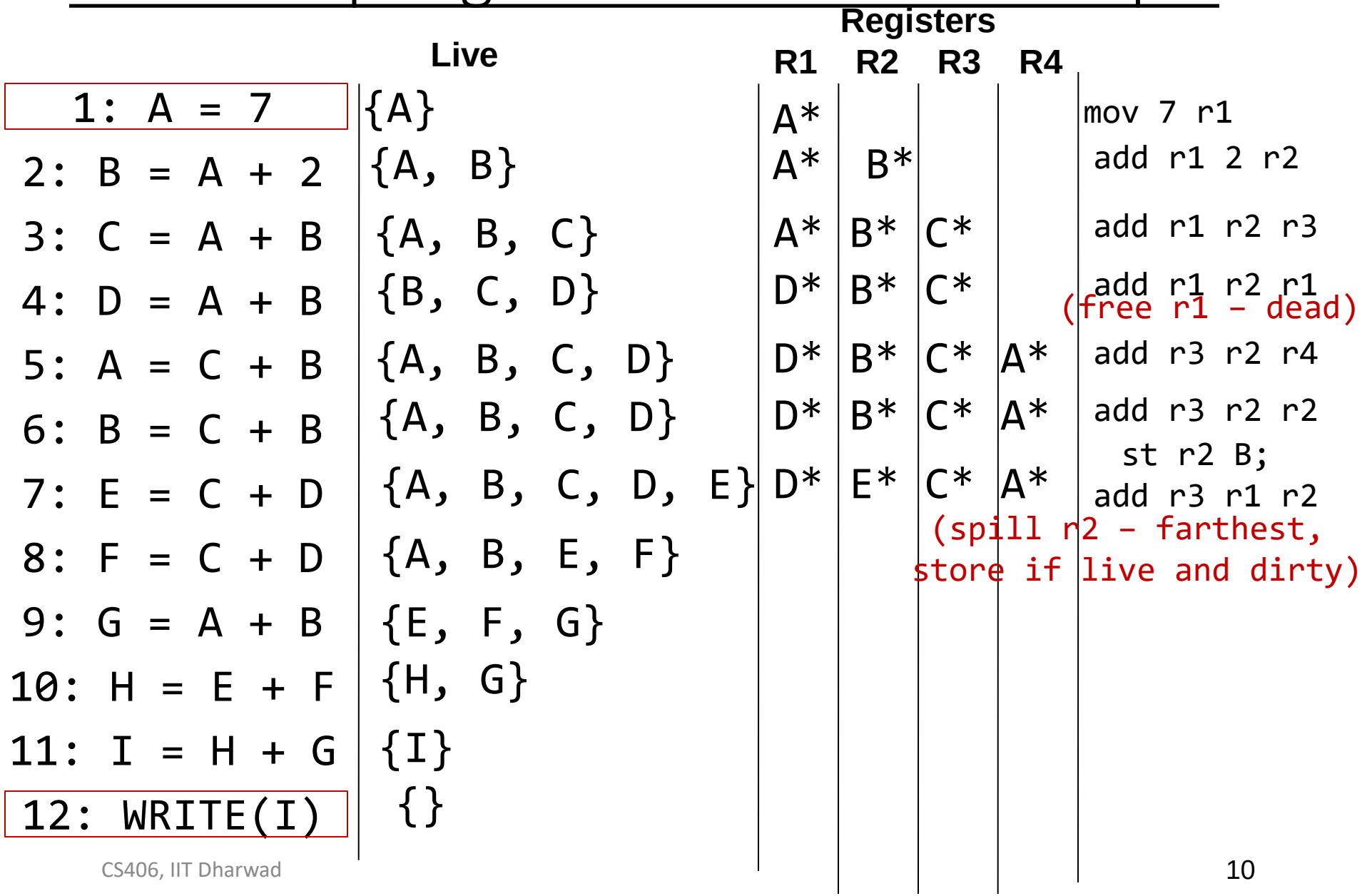
```
allocate(opr)
    if there is a free r
        choose r
    else
        choose r to free
        free(r)
    mark r associated with opr
    return r
```

Liveness Example

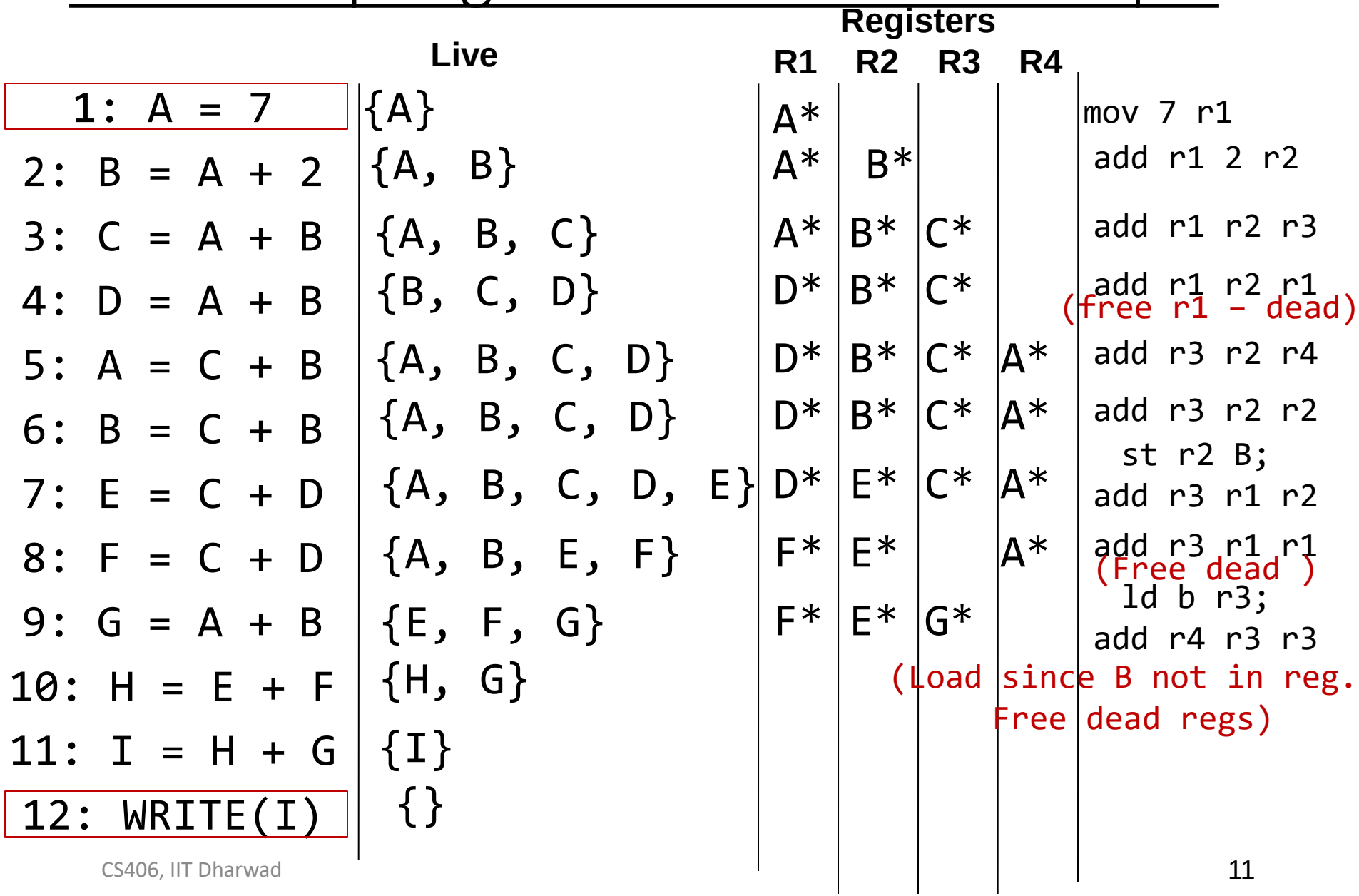
- What is live in this code? *Recall: a variable is live only if its value is used in future.*

	Live	Comments
1: A = B + C	{A, B}	Used B, C Killed A
2: C = A + B	{A, B, C}	Used A, B Killed C
3: T1 = B + C	{A, B, C, T1}	Used B, C Killed T1
4: T2 = T1 + C	{A, B, C, T2}	Used T1, C Killed T2
5: D = T2	{A, B, C, D}	Used T2, Killed D
6: E = A + B	{C, D, E}	Used A, B Killed E
7: B = E + D	{B, C, D}	Used E, D Killed B
8: A = C + D	{A, B}	Used C, D Killed A
9: T3 = A + B	{T3}	Used A, B Killed T3
10: WRITE(T3)	{}	Used T3

Bottom-up register allocation - Example



Bottom-up register allocation - Example



Bottom-up register allocation - Example

	Live	Registers R1	R2	R3	R4	
1: A = 7	{A}	A*				mov 7 r1
2: B = A + 2	{A, B}	A*	B*			add r1 2 r2
3: C = A + B	{A, B, C}	A*	B*	C*		add r1 r2 r3
4: D = A + B	{B, C, D}	D*	B*	C*		add r1 r2 r1 (free r1 – dead)
5: A = C + B	{A, B, C, D}	D*	B*	C*	A*	add r3 r2 r4
6: B = C + B	{A, B, C, D}	D*	B*	C*	A*	add r3 r2 r2 st r2 B;
7: E = C + D	{A, B, C, D, E}	D*	E*	C*	A*	add r3 r1 r2
8: F = C + D	{A, B, E, F}	F*	E*		A*	add r3 r1 r1 (Free dead) ld b r3;
9: G = A + B	{E, F, G}	F*	E*	G*		add r4 r3 r3
10: H = E + F	{H, G}	H*		G*		add r2 r1 r1
11: I = H + G	{I}	I*				add r1 r3 r1
12: WRITE(I)	{}					write r1

Instruction Scheduling

Instruction Scheduling

- Code generation has created a sequence of assembly instructions
- But that is not the only valid order in which instructions could be executed!

```
LD A, R1                    LD C, R4
LD B, R2                    LD B, R2
R3 = R1 + R2                LD A, R1
LD C, R4          ------>   R5 = R4 * R2
R5 = R4 * R2                R3 = R1 + R2
R6 = R3 + R5                R6 = R3 + R5
ST R6, D                    ST R6, D
```

- Different orders can give you better performance, more instruction level parallelism, etc.

Why do Instruction Scheduling?

- Not all instructions are the same
  - Loads tend to take longer than stores, multiplies tend to take longer than adds
- Hardware can overlap execution of instructions (pipelining)
  - Can do some work while waiting for a load to complete
- Hardware can execute multiple instructions at the same time (superscalar)
  - Hardware has multiple functional units

Why do Instruction Scheduling? Contd..

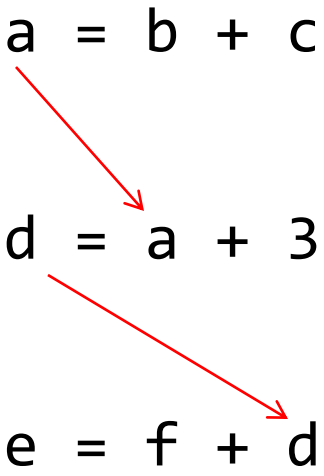
- VLIW (very long instruction word)
  - Popular in the 1990s, still common in some DSPs
  - Relies on compiler to find best schedule for instructions, manage instruction-level parallelism
  - Instruction scheduling is vital
- Out-of-order superscalar
  - Standard design for most CPUs (some low energy chips, like in phones, may be in-order)
  - Hardware does scheduling, but in limited window of instructions
  - Compiler scheduling still useful to make hardware's life easier

Instruction Scheduling - Considerations

- Gather constraints on schedule:
  - Data dependences between instructions
  - Resource constraints
- Schedule instructions while respecting constraints
  - List scheduling
  - Height-based heuristic

Data dependence constraints

- Are all instruction orders legal?



- Dependences between instructions prevent reordering

Data dependences

- Variables/registers defined in one instruction are used in a later instruction: **flow dependence**
- Variables/registers used in one instruction are overwritten by a later instruction: **anti dependence**
- Variables/registers defined in one instruction are overwritten by a later instruction: **output dependence**
- Data dependences prevent instructions from being reordered, or executed at the same time.

Other constraints

- Some architectures have more than one ALU

```
a = b * c
d = e + f
```

These instructions do not have any dependence. Can be executed in parallel

- But what if there is only one ALU?
  - Cannot execute in parallel
  - If a multiply takes two cycles to complete, cannot even execute the second instruction immediately after the first
- **Resource constraints** are limitations of the hardware that prevent instructions from executing at a certain time

Representing constraints

- **Dependence** constraints and **resource** constraints limit valid orders of instructions
- Instruction scheduling goal:
  - For each instruction in a program (basic block), assign it a *scheduling slot*
  - Which functional unit to execute on, and when
  - As long as we obey all of the constraints
- So how do we represent constraints?

Data dependence graph

- Graph that captures data dependence constraints
- Each node represents one instruction
- Each edge represents a dependence from one instruction to another
- Label edges with instruction *latency* (how long the first instruction takes to complete → how long we have to wait before scheduling the second instruction)

Example

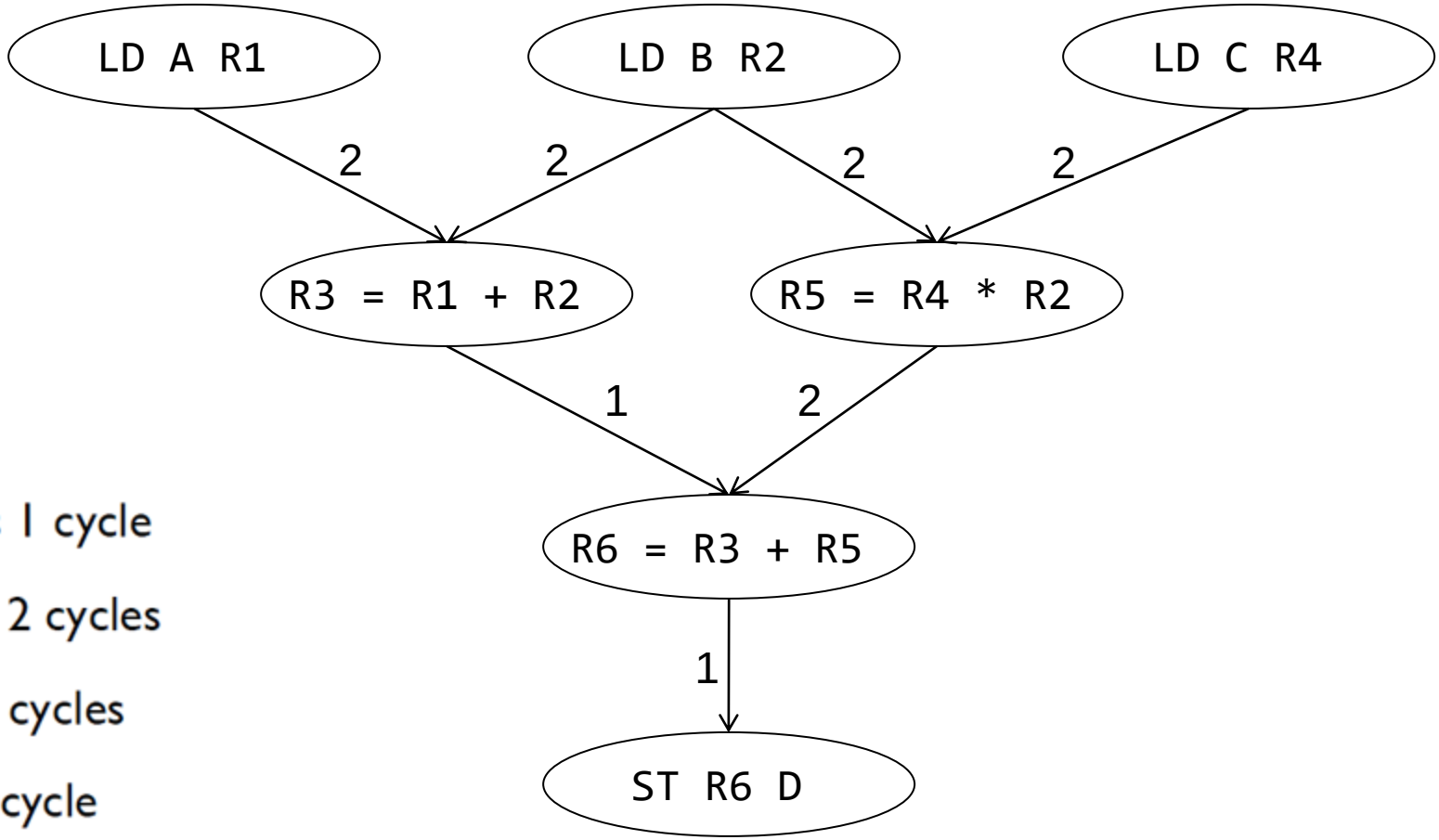
- ADD takes 1 cycle
- MUL takes 2 cycles
- LD takes 2 cycles
- ST takes 1 cycle

```
LD A, R1
LD B, R2
R3 = R1 + R2
LD C, R4
R5 = R4 * R2
R6 = R3 + R5
ST R6, D
```

Example

LD A, R1
LD B, R2
R3 = R1 + R2
LD C, R4
R5 = R4 * R2
R6 = R3 + R5
ST R6, D

- ADD takes 1 cycle
- MUL takes 2 cycles
- LD takes 2 cycles
- ST takes 1 cycle



Reservation tables

- Represent resource constraints using reservation tables
- For each instruction, table shows which functional units are occupied in each cycle the instruction executes
  - # rows: latency of instruction
  - # columns: number of functional units
  - T[i][j] marked ⇔ functional unit *j* occupied during cycle *i*
    - Caveat: some functional units are *pipelined*: instruction takes multiple cycles to complete, but only occupies the unit for the first cycle
- Some instructions have multiple ways they can execute: one table per variant

Example

- Two ALUs, fully pipelined
- One LD/ST unit, *not pipelined*
- ADDs can execute on ALU0 or ALU1
- MULs can execute on ALU0 only
- LOADs and STOREs both occupy the LD/ST unit

- ADD takes 1 cycle
- MUL takes 2 cycles
- LD takes 2 cycles
- ST takes 1 cycle

ALU0	ALU1	LD/ST

Example

- Two ALUs, fully pipelined
- One LD/ST unit, *not pipelined*

ALU0	ALU1	LD/ST

Example

- Two ALUs, fully pipelined
- One LD/ST unit, *not pipelined*
- ADDs can execute on ALU0 or ALU1

- ADD takes 1 cycle
- MUL takes 2 cycles
- LD takes 2 cycles
- ST takes 1 cycle

ALU0	ALU1	LD/ST
X		

ADD (1)

ALU0	ALU1	LD/ST
	X	

ADD (2)

Example

- Two ALUs, fully pipelined
- One LD/ST unit, *not pipelined*
- ADDs can execute on ALU0 or ALU1
- MULs can execute on ALU0 only

- ADD takes 1 cycle
- MUL takes 2 cycles
- LD takes 2 cycles
- ST takes 1 cycle

ALU0	ALU1	LD/ST
X		

MUL

Example

- Two ALUs, fully pipelined
- One LD/ST unit, *not pipelined*
- ADDs can execute on ALU0 or ALU1
- MULs can execute on ALU0 only
- LOADs and STOREs can execute on LD/ST unit only

- ADD takes 1 cycle
- MUL takes 2 cycles
- LD takes 2 cycles
- ST takes 1 cycle

ALU0	ALU1	LD/ST
		X

LOAD ?

What is incorrect here?

ALU0	ALU1	LD/ST
		X

STORE

Example

ADD(1)

ALU0	ALU1	LD/ST
X		

LOAD

ALU0	ALU1	LD/ST
		X
		X

ADD(2)

ALU0	ALU1	LD/ST
	X	

STORE

ALU0	ALU1	LD/ST
		X

MUL

ALU0	ALU1	LD/ST
X		

MUL still takes two cycles. Since ALU is fully pipelined, only occupies the ALU for 1

Can use reservation tables to see if instructions can be scheduled: see if tables overlap

Using tables

ADD(1)

ALU0	ALU1	LD/ST
X		

LOAD

ALU0	ALU1	LD/ST
		X
		X

ADD(2)

ALU0	ALU1	LD/ST
	X	

STORE

ALU0	ALU1	LD/ST
		X

MUL

ALU0	ALU1	LD/ST
X		

Which of the sequences below are valid?
| = run instructions in same cycle
; = move to next cycle

ADD | ADD ✓
ADD | MUL ✓
MUL | MUL ✗

MUL ; MUL | ADD ✓
LOAD | MUL ✓
LOAD ; STORE ✗

STORE ; LOAD ✓

Scheduling

- Can use these constraints to schedule a program
- Data dependence graph tells us what instructions are *available for scheduling* (have all of their dependences satisfied)
- Reservation tables help us build schedule by telling us which functional units are occupied in which cycle

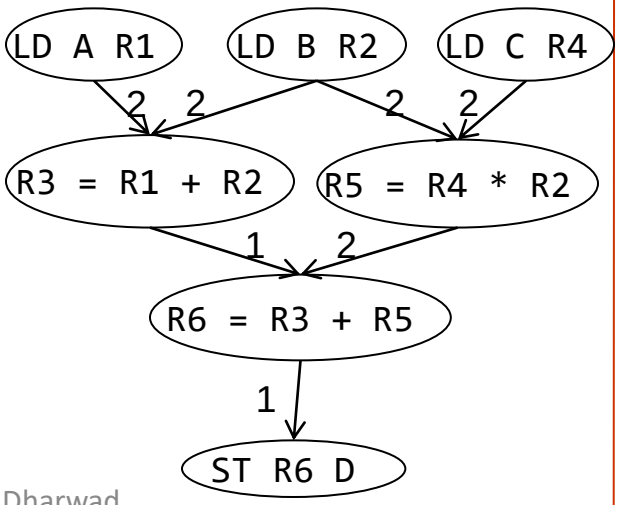
List scheduling

1. Start in cycle 0
2. For each cycle
    1. Determine which instructions are available to execute
    2. From list of instructions, pick one to schedule, and place in schedule
    3. If no more instructions can be scheduled, move to next cycle

Cycle	ALU0	ALU1	LD/ST
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

List scheduling - Example

```
1. LD A, R1
2. LD B, R2
3. R3 = R1 + R2
4. LD C, R4
5. R5 = R4 * R2
6. R6 = R3 + R5
7. ST R6, D
```



Cycle #	Available Instruction(s)	Scheduled Instruction(s)	Completed Instruction(s)
0	1, 2, 4	1*	
1	2, 4		
2	2, 4	2*	1
3	4		
4	3,4	3,4	2
5			3
6	5	5	4
7			
8	6	6	5
9	7	7	6
10			7

*an instruction from the list of available instructions is picked at random and scheduled

List scheduling

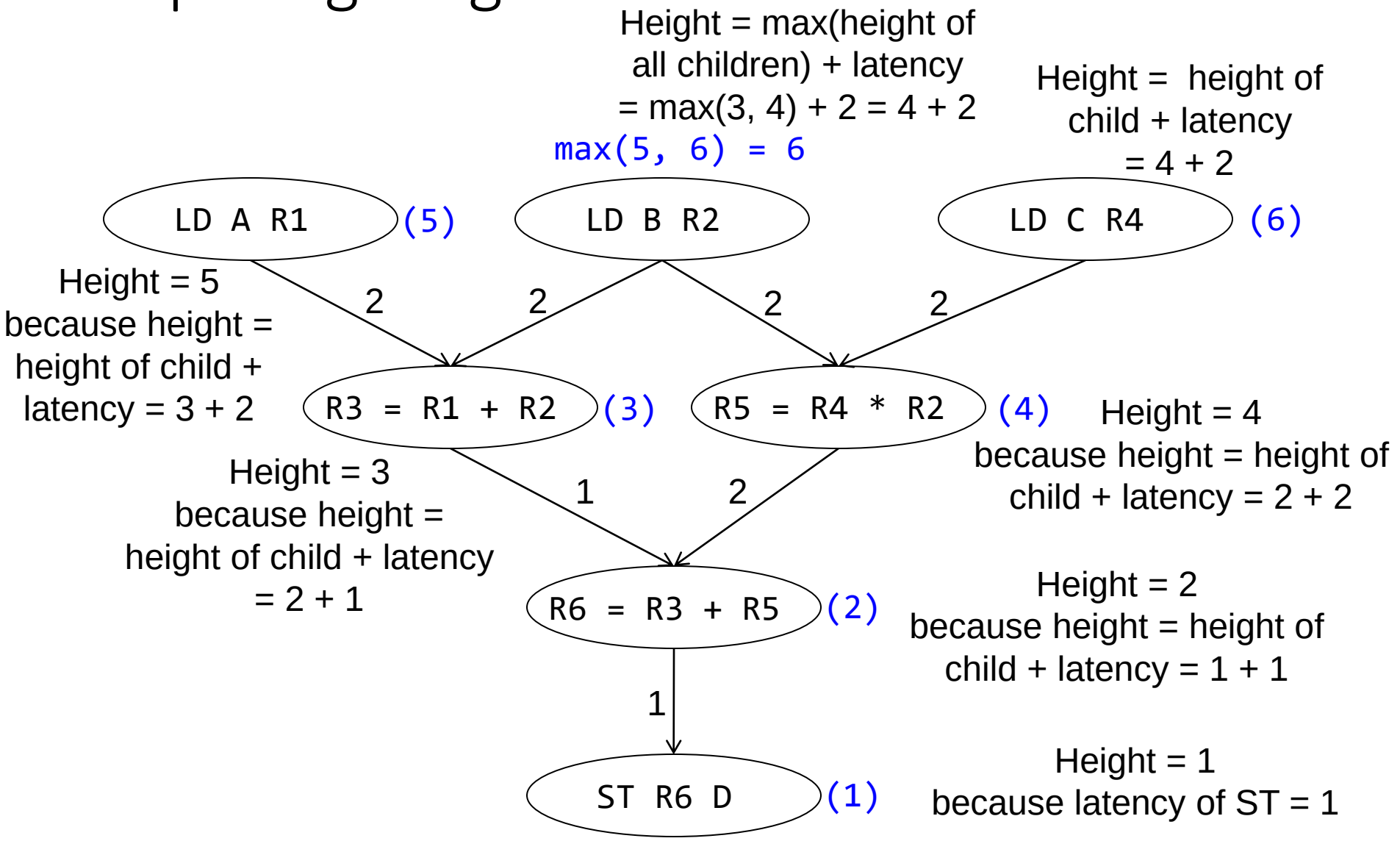
1. LD A, R1
2. LD B, R2
3. R3 = R1 + R2
4. LD C, R4
5. R5 = R4 * R2
6. R6 = R3 + R5
7. ST R6, D

Cycle	ALU0	ALU1	LD/ST
0			1
1			1
2			2
3			2
4	3		4
5			4
6	5		
7			
8	6		
9			7
10			

Height-based scheduling

- Important to prioritize instructions
  - Instructions that have a lot of downstream instructions dependent on them should be scheduled earlier
- Instruction scheduling NP-hard in general, but **height-based scheduling** is effective
- Instruction height = latency from instruction to farthest-away leaf
  - Leaf node height = instruction latency
  - Interior node height = max(heights of children + instruction latency)
- Schedule instructions with highest height first

Computing heights



Height-based list scheduling

1. LD A, R1
2. LD B, R2
3. R3 = R1 + R2
4. LD C, R4
5. R5 = R4 * R2
6. R6 = R3 + R5
7. ST R6, D

Cycle	ALU0	ALU1	LD/ST
0			2
1			2
2			4
3			4
4	5		1
5			1
6	3		
7	6		
8	7		
9			
10			