CS406: Compilers

Spring 2021

Week 10: Local Optimizations

(slide courtesy: Prof. Milind Kulkarni)

Naïve approach

- "Macro-expansion"
  - Treat each 3AC instruction separately, generate code in isolation

$$\text{ADD A, B, C} \longrightarrow \begin{array}{l}\text{LD A, R1}\\ \text{LD B, R2}\\ \text{ADD R1, R2, R3}\\ \text{ST R3, C}\end{array}$$

$$\text{MUL A, 4, B} \longrightarrow \begin{array}{l}\text{LD A, R1}\\ \text{MOV 4, R2}\\ \text{MUL R1, R2, R3}\\ \text{ST R3, B}\end{array}$$

Why is this bad? (I)

```
                      LD A, R1
                      MOV 4, R2
MUL A, 4, B  ------>  MUL R1, R2, R3
                      ST R3, B
```

```
                      LD A, R1
MUL A, 4, B  ------>  MULI R1, 4, R3
                      ST R3, B
```

Too many instructions
Should use a different instruction type

Why is this bad? (II)

```
ADD A, B, C    ───────►    LD A, R1
                           LD B, R2
                           ADD R1, R2, R3
                           ST R3, C
```

```
                           LD A, R1
                           LD B, R2
                           ADD R1, R2, R3
ADD A, B, C                ST R3, C
ADD C, A, E    ───────►    LD C, R4
                           LD A, R5
                           ADD R4, R5, R6
                           ST R6, E
```

Redundant load of C
Redundant load of A
Uses a lot of registers

Why is this bad? (III)

```
ADD A, B, C    ───────►    LD A, R1
                           LD B, R2
                           ADD R1, R2, R3
                           ST R3, C
```

```
                           ┌────────────────┐
                           │LD A, R1        │
                           │LD B, R2        │
                           │ADD R1, R2, R3  │
                           └────────────────┘
ADD A, B, C                ST R3, C
ADD A, B, D    ───────►    ┌────────────────┐
                           │LD A, R4        │
                           │LD B, R5        │
                           │ADD R4, R5, R6  │
                           └────────────────┘
                           ST R6, D
```

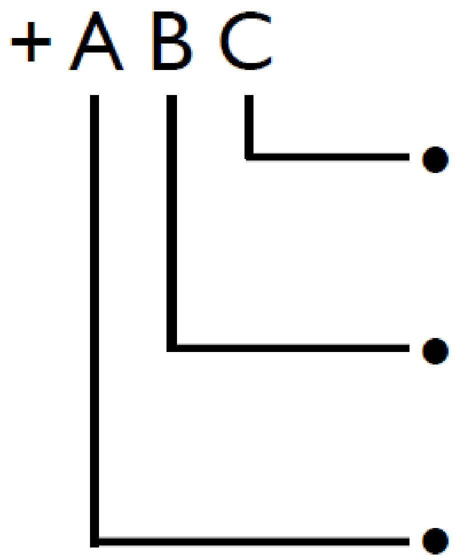
Wasting instructions recomputing A + B

How do we address this?

- Several techniques to improve performance of generated code
    - *Instruction selection* to choose better instructions
    - *Peephole optimizations* to remove redundant instructions
    - *Common subexpression elimination* to remove redundant computation
    - *Register allocation* to reduce number of registers used

Instruction selection

- Even a simple instruction may have a large set of possible address modes and combinations



+ A B C

- Can be indirect, register, memory address, indexed, etc.
- Can be literal, register, memory address, indexed, etc.
- Can be literal, register, memory address, indexed, etc.

- Dozens of potential combinations!

More choices for instructions

- Auto increment/decrement (especially common in embedded processors as in DSPs)
  - e.g., load from this address and increment it
  - Why is this useful?
- Three-address instructions
- Specialized registers (condition registers, floating point registers, etc.)
- "Free" addition in indexed mode

  MOV (R1)offset R2

  - Why is this useful?

Peephole optimizations

- Simple optimizations that can be performed by pattern matching
  - Intuitively, look through a "peephole" at a small segment of code and replace it with something better
  - Example: if code generator sees `ST R X; LD X R`, eliminate load
- Can recognize sequences of instructions that can be performed by single instructions

  `LDI R1 R2; ADD R1 4 R1` replaced by

  `LDINC R1 R2 4` //load from address in R1 then inc by 4

Peephole optimizations

- Simple optimizations that can be performed by pattern matching
    - Intuitively, look through a "peephole" at a small segment of code and replace it with something better
    - Example: if code generator sees `ST R X; LD X R`, eliminate load

Get the data present at address in R2 and put it in R1

be

`LDI R1 R2; ADD R1 4 R1` replaced by

`LDINC R1 R2 4` //load from address in R1 then inc by 4

Peephole optimizations

- Constant folding

```
ADD lit1, lit2, Rx  ⟶  MOV lit1 + lit2, Rx
MOV lit1, Rx
ADD li2, Rx, Ry     ⟶  MOV lit1 + lit2, Ry
```

- Strength reduction

```
MUL operand, 2, Rx  ⟶  SHIFTL operand, 1, Rx
DIV operand, 4, Rx  ⟶  SHIFTR operand, 2, Rx
```

- Null sequences

```
MUL operand, 1, Rx  ⟶  MOV operand, Rx
ADD operand, 0, Rx  ⟶  MOV operand, Rx
```

Peephole optimizations

- Combine operations

```
JEQ L1
JMP L2            ⟶  JNE L2
L1: ...
```

- Simplifying

```
SUB operand, 0, Rx  ⟶  NEG Rx
```

- Special cases (taking advantage of ++/--)

```
ADD 1, Rx, Rx       ⟶  INC Rx
SUB Rx, 1, Rx       ⟶  DEC Rx
```

- Address mode operations

```
MOV A R1
ADD 0(R1) R2 R3     ⟶  ADD @A R2 R3
```

Superoptimization

- Peephole optimization/instruction selection writ large
- Given a sequence of instructions, find a different sequence of instructions that performs the same computation in less time
- Huge body of research, pulling in ideas from all across computer science
    - Theorem proving
    - Machine learning

Common subexpression elimination

- Goal: remove redundant computation, don't calculate the same expression multiple times

```
1: A = B * C
2: E = B * C
```

Keep the result of statement 1 in a temporary and reuse for statement 2

- Difficulty: how do we know when the same expression will produce the same result?

```
1: A = B * C
2: B = <new value>
3: E = B * C
```

B is "killed." Any expression using B is no longer "available," so we cannot reuse the result of statement 1 for statement 3

- This becomes harder with pointers (how do we know when B is killed?)

Common subexpression elimination

- Two varieties of common subexpression elimination (CSE)
- Local: within a single basic block
    - Easier problem to solve (why?)
- Global: within a single procedure or across the whole program
    - Intra- vs. inter-procedural
    - More powerful, but harder (why?)
    - Will come back to these sorts of “global” optimizations later

CSE in practice

- Idea: keep track of which expressions are "available" during the execution of a basic block
    - Which expressions have we already computed?
    - Issue: determining when an expression is no longer available
        - This happens when one of its components is assigned to, or "killed."
- Idea: when we see an expression that is already available, rather than generating code, copy the temporary
    - Issue: determining when two expressions are the same

Maintaining available expressions

- For each 3AC operation in a basic block
    - Create name for expression (based on lexical representation)
    - If name not in available expression set, generate code, add it to set
        - Track register that holds result of and any variables used to compute expression
    - If name in available expression set, generate move instruction
    - If operation assigns to a variable, kill all dependent expressions

Example

3 Address Code	Available expression(s)	Killed expression(s)	Generated Code (assembly)
ADD A B T1	{}		add a b r1
ADD T1 C T2	{“A + B”}		add r1 c r2
ADD A B T3	{“A + B”, “T1 + C”}		mov r1 r3
ADD T1 T2 C	{“A + B”, ~~“T1 + C”}~~	{“T1+C”}	add r1 r2 r5 st r5 c
ADD T1 C T4	{“A + B”, “T1 + T2”}		add r1 c r4
ADD T3 T2 D	{“A + B”, “T1 + T2”, “T1 + C”}		add r3 r2 r6 st r6 d
	{“A + B”, “T1 + T2”, “T1 + C”, “T3 + T2”}		

Downsides (CSE)

- What are some downsides to this approach? Consider the two highlighted operations

Three address code

```
+ A B T1
+ T1 C T2
+ A B T3
+ T1 T2 C
+ T1 C T4
+ T3 T2 D
```

Generated code

```
ADD A B R1
ADD R1 C R2
MOV R1 R3
ADD R1 R2 R5; ST R5 C
ADD R1 C R4
ST R5 D
```

T1 and T3 compute the same expression. This can be handled by an optimization called *value numbering.*

Aliasing

- One of the biggest problems in compiler analysis is to recognize aliases – different names for the same location in memory

exercise: are T1 and T3 aliased in previous example?

- Why do aliases occur?
  - Pointers referring to the same location
  - Function calls passing the same reference in two arguments
  - Arrays referencing the same element
  - Unions
- What problems does aliasing pose for CSE?
  - when talking about “live” and “killed” values in optimizations like CSE, we’re talking about particular variable names
  - In the presence of aliasing, we may not know which variables get killed when a location is written to

Memory disambiguation

- Most compiler analyses rely on *memory disambiguation*
  - Otherwise, they need to be too conservative and are not useful
- Memory disambiguation is the problem of determining whether two references point to the same memory location
  - *Points-to* and *alias* analyses try to solve this
  - Will cover basic pointer analyses in a later lecture

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 (but not inter-procedural – why?)

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

Liveness Example

- What is live in this code?

	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

```
For each tuple op A B C in a BB, do
    $R_x$ = ensure(A)
    $R_y$ = ensure(B)
    if A *dead* after this tuple, free($R_x$)
    if B *dead* after this tuple, free($R_y$)
    $R_z$ = allocate(C) //could use $R_x$ or $R_y$
    generate code for op
    mark $R_z$ *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
```

Bottom-up register allocation - Example

	Live	R1	R2	R3	R4	
		Registers				
1: A = 7	{A}	A*				mov 7 A
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
7: E = C + D	{A, B, C, D, E}	D*	E*	C*	A*	st r2 B; add r3 r1 r2 (spill r2 – farthest, store if live and dirty)
8: F = C + D	{A, B, E, F}	F*	E*		A*	add r1 r3 r1 (Free dead)
9: G = A + B	{E, F, G}	F*	E*	G*		ld b r3; add r4 r3 r3
10: H = E + F	{H, G}	H*		G*		add r1 r2 r1 (Load since B not in reg. Free dead regs)
11: I = H + G	{I}	I*				add r1 r3 r1
12: WRITE(I)	{}					write r1

Exercise

Do bottom-up register allocation with 3 registers. When choosing a register to allocate always choose the lowest numbered one available. When choosing register to spill, choose the non-dirty register that will be used farthest in future. If all registers are dirty, choose the one that is used farthest in future. In case of a tie, choose the lowest numbered register.

```
A = B + C
C = A + B
T1 = B + C
T2 = T1 + C
D = T2
E = A + B
B = E + D
A = C + D
T3 = A + B
WRITE(T3)
```

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

$\longrightarrow$

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

How to do Instruction Scheduling?

- Consider 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?

$$\begin{tikzcd}
a \arrow[drr] & = & b & + & c \\
d \arrow[drrrr] & = & a & + & 3 \\
e & = & f & + & d
\end{tikzcd}$$

- 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	These instructions do not have any dependence. Can be executed in parallel
d = e + f	

- 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

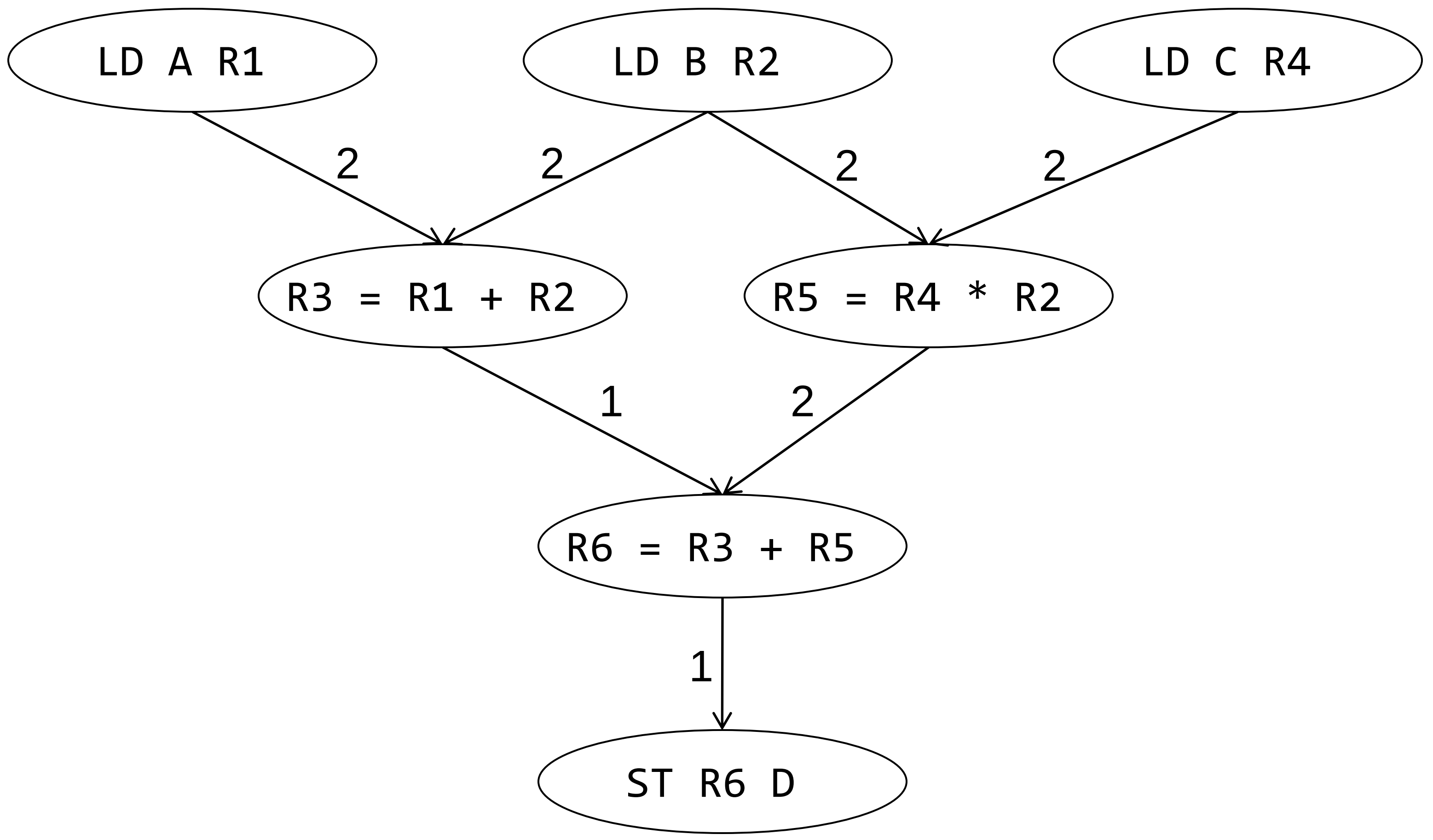
- Graph that captures data dependence constraints
- Each node represents one instruction
- Each edge represents a dependence from one instruction to another
- Label edges with instruction *latency* (how long the first instruction takes to complete → how long we have to wait before scheduling the second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



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

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

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

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

ALU0	ALU1	LD/ST
		X

LOAD

ALU0	ALU1	LD/ST
		X

STORE

Example

ADD(1)

ALU0	ALU1	LD/ST
X		

ADD(2)

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

LOAD

ALU0	ALU1	LD/ST
		X
		X

STORE

ALU0	ALU1	LD/ST
		X

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

Using tables

	ALU0	ALU1	LD/ST
ADD(1)	X		

	ALU0	ALU1	LD/ST
LOAD			X
			X

	ALU0	ALU1	LD/ST
ADD(2)		X	

	ALU0	ALU1	LD/ST
STORE			X

	ALU0	ALU1	LD/ST
MUL	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

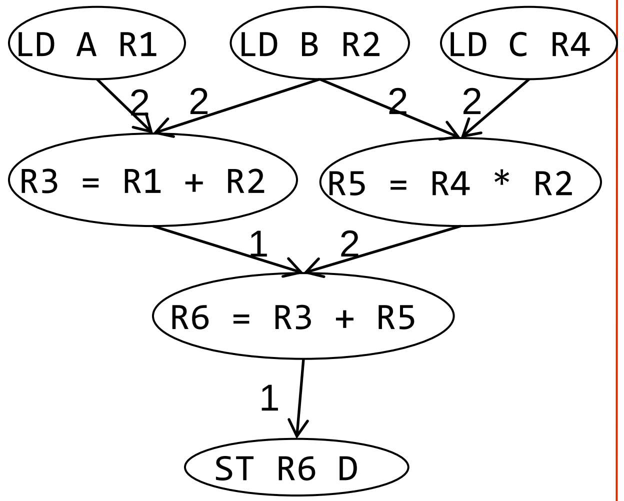
List scheduling

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			

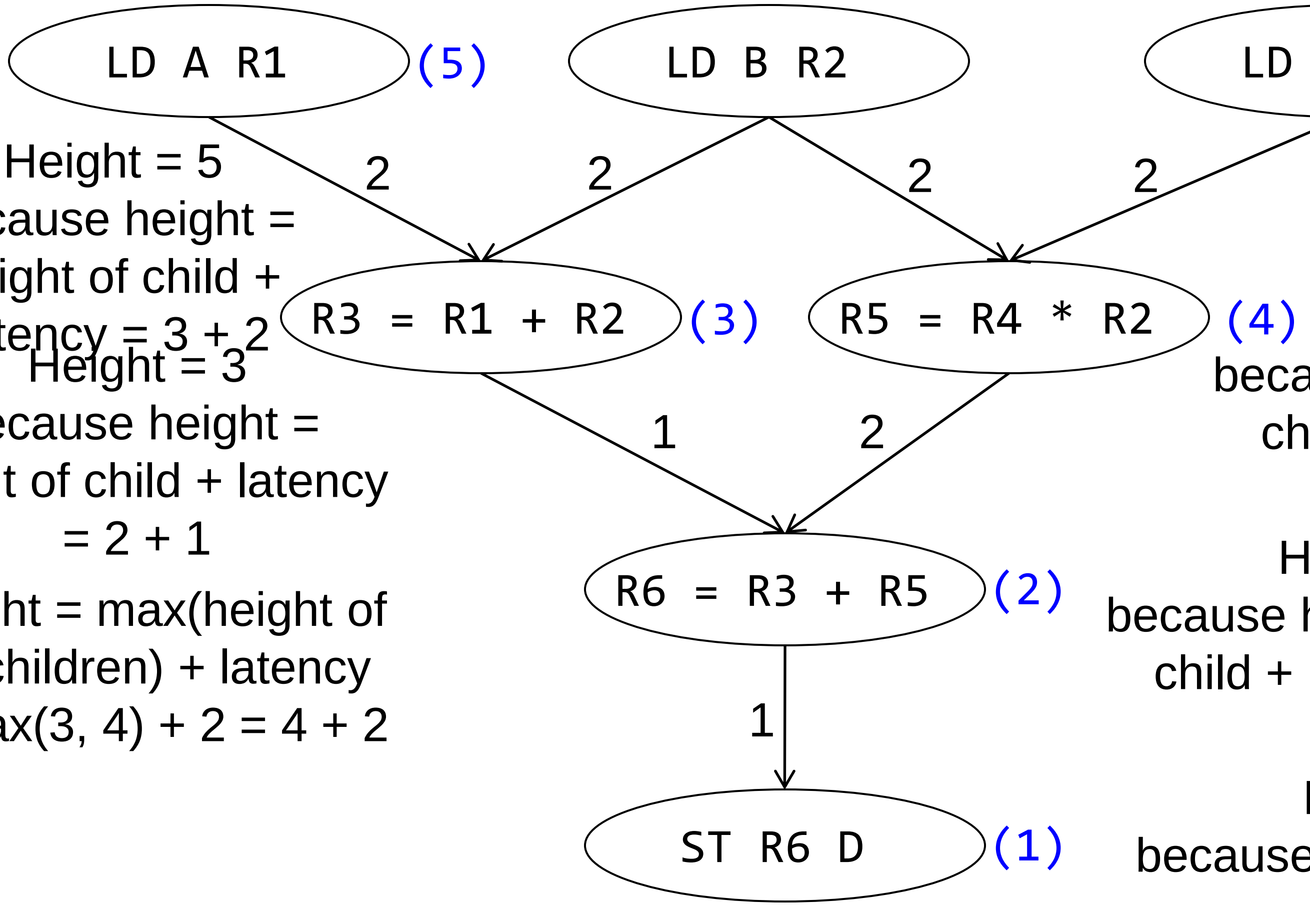
Height-based scheduling

- 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 = height of child + latency = 4 + 2

max(5, 6) = 6



Height = 5 because height = height of child + latency = 3 + 2

Height = 2 because height = height of child + latency = 2 + 2

Height = 3 because height = height of child + latency = 2 + 1

Height = max(height of all children) + latency = max(3, 4) + 2 = 4 + 2

Height = 2 because height = height of child + latency = 1 + 1

Height = 1 because latency of ST = 1

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			

Instruction Scheduling - Exercise

- 2 ALUs (fully pipelined) and one LD/ST unit (not pipelined) are available.
- Either of the ALUs can execute ADD (1 cycle). Only one of the ALUs can execute MUL (2 cycles).
- LDs take up an ALU for 1 cycle and LD/ST unit for two cycles.
- STs take up an ALU for 1 cycle and LD/ST unit for one cycle.

i) Draw reservation tables, ii)DAG for the code shown iii) schedule using height based list scheduling.

```
1: LD A R1
2: LD B R2
3: LD C R3
4: LD D R4
5: R5 = R1 + R2
6: R6 = R5 * R3
7: R7 = R1 + R6
8: R8 = R6 + R5
9: R9 = R4 + R7
10: R10 = R9 + R8
11: ST R10  E
12: ST R7  F
```