Instruction scheduling

What is 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

Different types of hardware

- 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

Outline

- Constraints on schedule
  - Dependences between instructions
  - Resource constraints
- Scheduling instructions while respecting constraints
  - List scheduling
  - Height-based heuristic

Scheduling constraints

- Are all instruction orders legal?

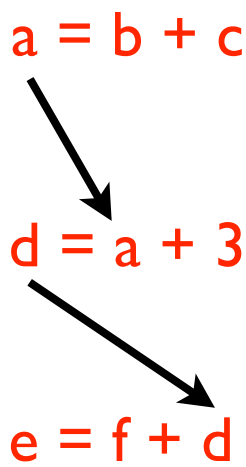
a = b + c

d = a + 3

e = f + d

Scheduling 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
```

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

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

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		

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

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

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

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			
2			
3			
4			
5			
6			
7			
8			
9			
10			

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

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			