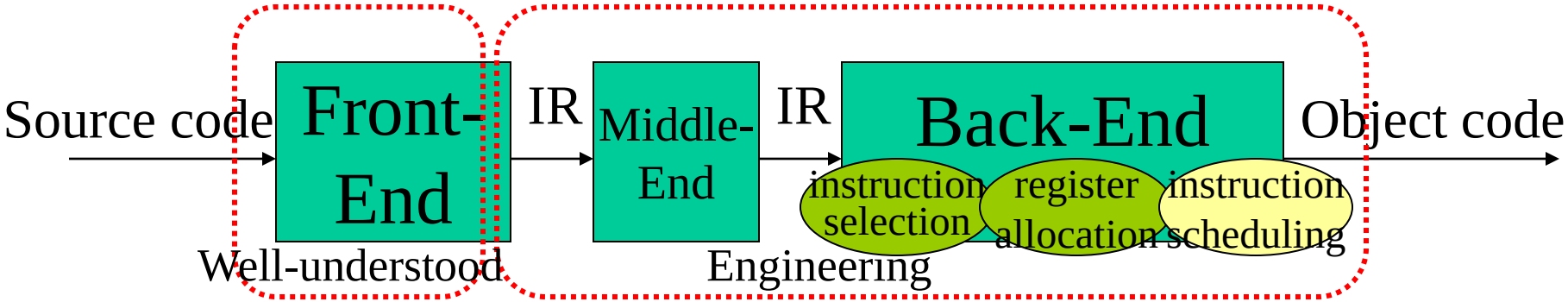
Instruction Scheduling



- Instruction Scheduling
  - The problem:
    - Given a code fragment for some target machine and the latencies for each individual operation, reorder operations to minimise execution time.
    - Recall: modern processors may have multiple functional units.
  - The task:
    - Produce correct code; minimise wasted cycles; avoid spilling registers; operate efficiently.

Background

- Many operations have delay latencies for execution.
  - E.g., load, store: <delay> CPU cycles (depends on the processor)
    - Issue load, result appears <delay> cycles later.
    - Execution continues unless result is referenced.
    - Premature reference causes hardware to stall.
- Modern machines can issue several operations per cycle.
- Execution time is order-dependent (has been since the 60s)
- Overview of a solution:
  - Move loads back at least <delay> slots from where they are needed, but this increases register pressure (i.e., more registers may be needed) – recall the example in lecture 14 (slide 7).
    Ideally, we want to minimise both hardware stalls and added register pressure.

Motivating Example

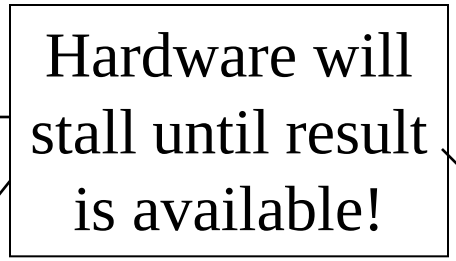
- Two variants to compute (a+b)+c:

(9 cycles):

```
load r1, @a
load r2, @b
add r1,r1,r2
load r3, @c
add r1,r1,r3
```

(6 cycles):

```
load r1, @a
load r2, @b
load r3, @c
add r1,r1,r2
add r1,r1,r3
```



(assume that the latency of a load is 3 cycles; all other instructions have a latency of 1 cycle)

(NB: costs due to the memory hierarchy are not part of the picture)

Instruction Scheduling for a basic block
The big picture

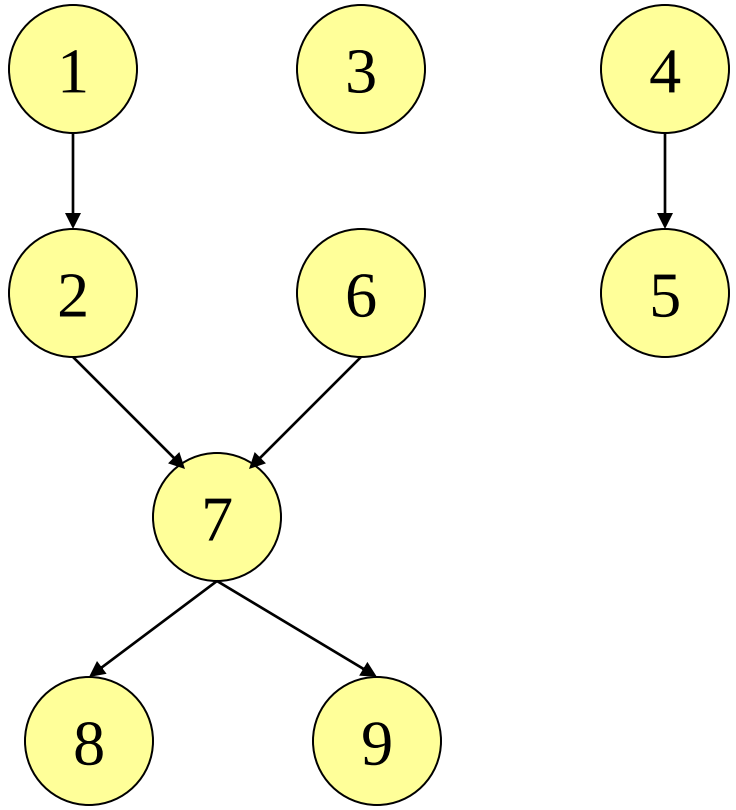
1. Build a **precedence** (data dependence) **graph**.
2. Compute a **priority function** for the nodes of the graph.
3. Use **list scheduling** to construct a schedule, one cycle at a time:
    1. Use a queue of operations that are ready
    2. At each cycle:
        1. Choose a ready operation and schedule it
        2. Update the ready queue.

A greedy heuristic; open to variations.

(greedy heuristic: An algorithmic technique in which an optimisation problem is solved by finding locally optimal solutions)

Build a precedence graph

1. load r1, @x
2. load r2, [r1+4]
3. and r3, r3, 0x00FF
4. mult r6, r6, 100
5. store r6
6. div r5, r5, 100
7. add r4, r2, r5
8. mult r5, r2, r4
9. store r4

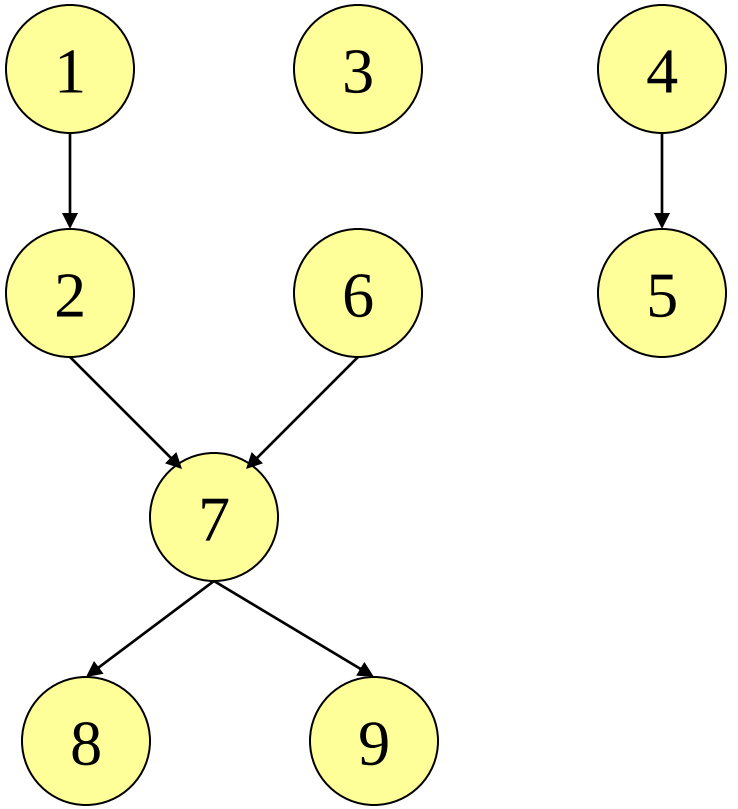


Compute a priority function

- Assign to each node a weight equal to the longest delay latency (total) to reach a leaf in the graph from this node (include latency of current node).
- $weight_i = latency_i + max(weight_{\text{all successor nodes}})$

(Assume: div 4 cycles, and mult 3 cycles; 1 cycle for the rest):

node	weight
1	6
2	5
3	1
4	4
5	1
6	8
7	4
8	3
9	1



(Local) List scheduling

```
Cycle=1; Ready=set of available operations; Active={}
While (Ready ∪ Active != {})
    if (Ready!={}) then
        remove an op from Ready (based on the weight)
        schedule(op)=cycle
        Active=Active ∪ op
    cycle=cycle+1
    for each op in Active
        if (schedule(op)+delay(op)<=cycle) then
            remove op from Active
            for each immediate successor s of op
                if (all operand of s are available) then
                    Ready=Ready ∪ s
```

Find the schedule

cycle	Instructions ready	Schedule	Instructions active
1	6, 1, 4, 3	div r5,r5,100	6
2	1, 4, 3	load r1,@x	6, 1
3	2, 4, 3	load r2, [r1+4]	6, 2
4	4, 3	mult r6,r6,100	6, 4
5	7, 3	add r4,r2,r5	4, 7
6	8, 3, 9	mult r5,r2,r4	4, 8
7	3, 9, 5	and r3,r3,0x00ff	8, 3
8	9, 5	store r4	8, 9
9	5	store r6	5

More list scheduling

- Two distinct classes of list scheduling:
  - Forward list scheduling: start with all available operations; work forward in time (Ready: all operands available)
  - Backward list scheduling: start with leaves; work backward in time (Ready: latency covers uses)
  - Folk wisdom is to try both and keep the best result.
- Variations on computing priority function:
  - Maximum path length containing node (decreases register usage).
  - Prioritise critical path.
  - Number of immediate successors or total number of descendants.
  - Increment weight if node contains a last use (shortens live ranges)
  - Do not add latency to node's weight.
  - Maximum delay latency from first available node.

Multiple functional units

- Modern architectures can run operations in parallel.
- List scheduling needs to be modified so that it can schedule as many operations per cycle as functional units (assuming that there are instructions available)

  (3rd line in the algorithm of slide 7 will have to be modified to:
  while (Ready!={} && there_are_free_functional_units)

- Back to the previous example (assume two functional units that can issue any instruction) [3rd column shows the ready set]

6. div r5, r5, 100	1. load r1, @x	{6, 1, 4, 3}
2. load r2, [r1+4]	4. mult r6, r6, 100	{2, 4, 3}
3. and r3, r3, 0x00FF	nop	{3}
nop	nop	{}
7. add r4, r2, r5	5. store r6	{7, 5}
8. mult r5, r2, r4	9. store r4	{8, 9}

Exercise

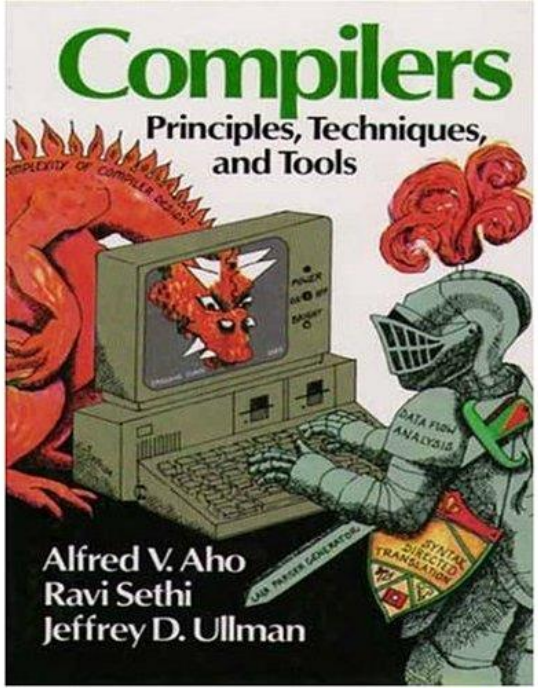
Assume two functional units: one for ALU operations only and another for memory operations only. A load and a mult have a latency of 2 cycles, all other instructions have a latency of 1 cycle.

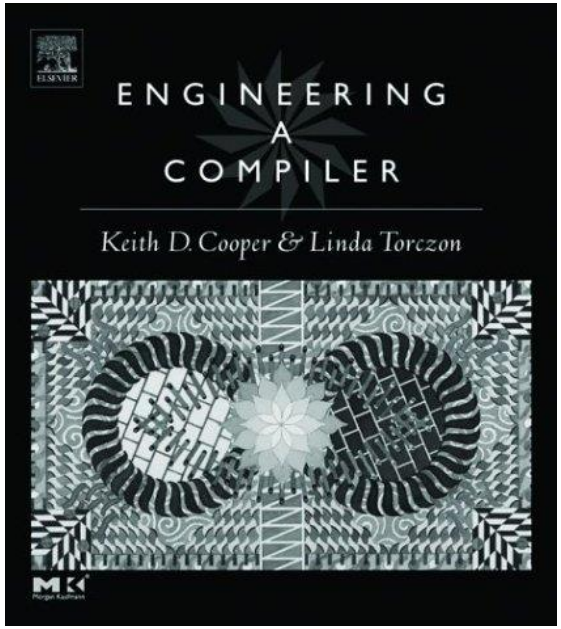
1. load r1, @x
2. load r2, @y
3. add r2,r2,42
4. load r3, @z
5. shl r4,r1,4
6. store r4
7. mult r5,r2,r3
8. add r6,r5,r4
9. store r5

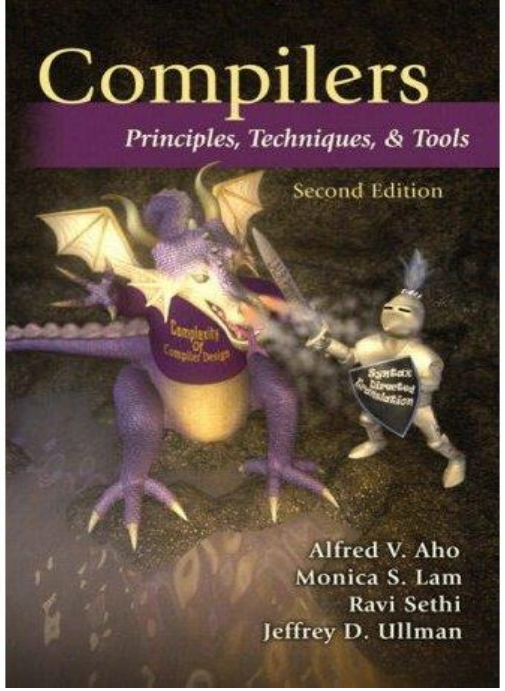
ALU	MEM
nop	2
nop	4
3	1
7	nop
5	nop
8	9
nop	6

Further Issues and Conclusion

- Going beyond basic blocks:
  - Identify high-frequency path and schedule as if a single block.
- Modulo scheduling:
  - Schedule multiple iterations together (and run several iterations concurrently - i.e., overlap successive iterations).
- Register allocation with instruction scheduling:
  - The former before the latter restricts the choices for scheduling.
  - If the latter before the former and register allocation has to spill registers, the whole (carefully done) schedule changes!
- Besides performance, we may want to minimise power consumption, size of the code, ...
- Most of the active compiler research revolves around these areas!
- Reading: Cooper, Chapter 12







References

- **Aho, Lam, Sethi, Ullman. "Compilers: Principles, Techniques and Tools", 2nd edition**. (Aho2) The **1st edition** (by Aho, Sethi, Ullman – Aho1), the "Dragon Book", has been a classic for over 20 years.
- **Cooper & Torczon. "Engineering a Compiler"** – an earlier draft has been consulted when preparing this module. (Cooper)
- Based on the material from: http://studentnet.cs.manchester.ac.uk/ugt/COMP36512/