The Back End:

1. Maps infinite number of virtual registers to finite number of real registers → register allocation
2. Removes inefficiencies introduced by front-end → optimizer
3. Removes inefficiencies introduced by programmer → optimizer
4. Adjusts pseudo-assembly composition and order to match target machine → scheduler

Scheduling

Multiply instruction takes 2 cycles...

1  r1 = r0 + 0
2  r2 = M[FP + A]
3  r3 = r0 + 4
4  r4 = M[FP + X]

LOOP:
1  r5 = r3 * r1
2  
3  r5 = r2 + r5
4  M[r5] = r4
5  r1 = r1 + 1
6  BR r1 <= 10, LOOP

1  r1 = r0 + 0
2  r2 = M[FP + A]
3  r3 = r0 + 4
4  r4 = M[FP + X]

LOOP:
1  r5 = r3 * r1
2  r1 = r1 + 1
3  r5 = r2 + r5
4  M[r5] = r4
5  BR r1 <= 10, LOOP
Multiply instruction takes 2 cycles...
Machine executes 2 instructions per cycle...

1 r1 = r0 + 0
2 r2 = M[FP + A]
3 r3 = r0 + 4
4 r4 = M[FP + X]

LOOP:
1 r5 = r3 * r1
2 r1 = r1 + 1
3 r5 = r2 + r5
4 M[r5] = r4
5 BR r1 <= 10, LOOP

Instruction Level Parallelism

- Instruction-Level Parallelism (ILP), the concurrent execution of independent assembly instructions.
- ILP is a cost effective way to extract performance from programs.
- Exploiting ILP requires global optimization and scheduling.
- Processors are becoming increasingly dependent on the ability of compilers to expose ILP.
  - Current state-of-the-art machines can execute 3 to 6 instructions per cycle if available. (i.e. Pentium III, DEC Alpha 21264)
  - Some processors rely on compiler for guidance. (i.e. Itanium)
- Current state-of-the-art compilers cannot expose this level of ILP in integer programs.

Data Dependence

- A data dependence is a constraint on scheduling arising from the flow of data between two instructions. Types:
  - RAW: An instruction \( u \) is flow-dependent on a preceding instruction \( d \) if \( u \) consumes a value computed by \( d \).
  - WAR: An instruction \( d \) is anti-dependent on a preceding instruction \( u \) if \( d \) writes to a location read by \( u \).
  - WAW: An instruction \( d_2 \) is output-dependent on a preceding instruction \( d_1 \) if \( d_1 \) writes to a location also written by \( d_2 \).
- Types of data:
  - Register dependence
  - Memory dependence
**Data Dependence**

\[ r_1 = r_2 + r_3 \]

\[ \text{Branch } r_1 \ <= \ 10, \ \text{TRUE} \]

\[ r_4 = r_2 \ast r_5 \]

\[ r_5 = r_4 + 1 \]

**FALSE:**

\[ r_4 = r_5 - 1 \]

---

**False Dependence**

 Eliminate WAW dependences

\[ r_1 = \]

\[ \text{branch} \]

\[ r_1 = \]

\[ = r_1 \]

Eliminate WAR dependences

\[ = r_1 \]

\[ r_1 = \]

\[ = r_1 \]

• Eliminate RAW dependences?
• Register allocation vs. splitting live ranges

---

**Control Dependence**

• A *control dependence* is a constraint on scheduling arising from the control flow of the program.

\[ \text{Branch } r_1 \ <= \ 10, \ \text{TARGET1} \]

\[ \text{Branch } r_2 \ <= \ 10, \ \text{TARGET2} \]

\[ r_4 = r_3 + 5 \]

**TARGET1:**

\[ r_5 = r_4 - 1 \]

**TARGET2:** (Assume: \( r_4 \) not live here)
Sources of Control Dependence

- Liveness
- Side-effects
  - Potentially Excepting Instructions (PEIs)
  - Memory Writes
  - Input/Output

Dependences

Latency
- Amount of time after the execution of an instruction that its result is ready.
- An instruction can have more than one latency!

Data Dependence Graph
- A data dependence graph consists of instructions and a set of directed data dependence edges among them in which each edge is labeled with its latency and type of dependence.
- Scheduling (code motion) must respect dependence graph.

Resources

- What does “two instructions per cycle” mean?
- Resource - A function of the processor that can be used by only one instruction at a time.
- Examples:
  - Fetch units
  - Decode units
  - Execution units
  - Register ports
Pipelining

Resource Map

Scheduling

- The goal of scheduling is to construct a sort of the dependence graph that:
  - Produces the same result - respects dependences
  - Minimizes execution time - makes maximal use of machine resources
- Scheduling is NP-hard even with simple formulation of problem.
- Use Heuristics to approximate solution.
- In practice, is exhaustive search of all schedules practical in most cases?
- List scheduling, the most common heuristic, is $O(n^2)$.
- Create `ready queue` to hold `ready` instructions.
- An instruction is `ready` when all incoming dependences are satisfied.
- A dependence is satisfied when source of dependence are has been scheduled at least `latency` cycles earlier.

---

**List Scheduling**

build dependence graph
insert instructions with no incoming dependences into ready queue
WHILE (instruction are not scheduled) DO
    `current_cycle.sched` = FALSE
    FOREACH instruction $i$ in ready queue DO
        IF (resources exist to schedule $i$ in cycle) THEN
            schedule $i$, update ready queue
            `current_cycle.sched` = TRUE
        ELSE
            current_cycle.sched = FALSE
            schedule $i$, update ready queue
    ENDFOREACH
ENDWHILE

---

**List Scheduling**

```plaintext
LOOP:
1  r5 = r3 * r1

2  r1 = r1 + 1

3  r5 = r2 + r5

4  M[r5] = r4

5  BR r1 <= 10, LOOP
```
Hardware Scheduling

Machines can also do scheduling...

- hardware schedulers process code after it has been fetched
- hardware finds independent instructions
- works with legacy architectures (found in x86, Pentium)
- program knowledge more precise at run-time - memory dependence

But compiler still important.

- Hardware schedulers have a small window.
- Hardware complexity increases.
- Hardware does not benefit directly from compiler optimization.

Expression Reformulation
Loop Unrolling

```
sum = 0;
for i = 1 to 30:
    sum = sum + A[i];

0  r1 = 0        r2 = 0
Loop:
0  r3 = M[r1 + A]  r1 = r1 + 1
1
2  r2 = r2 + r3   BR r1 < 30, Loop
```

Renaming

```
0  r1 = 0        r2 = 0
Loop:
0  r3 = M[r1 + A]  r1 = r1 + 1
1
2  r2 = r2 + r3
3  r3 = M[r1 + A]  r1 = r1 + 1
4
5  r2 = r2 + r3   BR r1 < 30, Loop
```

Accumulator Expansion

```
0  r1 = 0        r2 = 0
Loop:
0  r3 = M[r1 + A]  r1 = r1 + 1
1  r4 = M[r1 + A]  r1 = r1 + 1
2  r2 = r2 + r3
3  r2 = r2 + r4   BR r1 < 30, Loop
```
Accumulator Expansion

\[ \begin{align*}
0 & \quad r_1 = 0 \quad r_2 = 0 \\
\text{Loop:} & \\
0 & \quad r_3 = M[r_1 + A] \quad r_1 = r_1 + 1 \\
1 & \quad r_4 = M[r_1 + A] \quad r_1 = r_1 + 1 \\
2 & \quad r_5 = M[r_1 + A] \quad r_1 = r_1 + 1 \quad r_2 = r_2 + r_3 \\
3 & \quad r_2 = r_2 + r_4 \\
4 & \quad r_2 = r_2 + r_5 \quad \text{BR} \quad r_1 < 30, \text{Loop}
\end{align*} \]

Induction Variable Elimination

\[ \begin{align*}
0 & \quad r_1 = 0 \quad r_{23} = 0 \\
1 & \quad r_{24} = 0 \quad r_{25} = 0 \\
\text{Loop:} & \\
0 & \quad r_3 = M[r_1 + A] \quad r_1 = r_1 + 1 \\
1 & \quad r_4 = M[r_1 + A] \quad r_1 = r_1 + 1 \\
2 & \quad r_5 = M[r_1 + A] \quad r_1 = r_1 + 1 \quad r_{23} = r_{23} + r_3 \\
3 & \quad r_{24} = r_{24} + r_4 \\
5 & \quad r_{25} = r_{25} + r_5 \quad \text{BR} \quad r_1 < 30, \text{Loop} \\
0 & \quad r_2 = r_{23} + r_{24} \\
1 & \quad r_2 = r_2 + r_{25}
\end{align*} \]

Loop Unrolling and Optimization

\[ \begin{align*}
0 & \quad r_{13} = 0 \quad r_{14} = 1 \\
1 & \quad r_{15} = 2 \quad r_{23} = 0 \\
2 & \quad r_{24} = 0 \quad r_{25} = 0 \\
\text{Loop:} & \\
0 & \quad r_3 = M[r_{13} + A] \quad r_{13} = r_{13} + 3 \quad r_4 = M[r_{14} + A] \\
 & \quad r_{14} - r_{14} + 3 \quad r_{13} - M[r_{15} + A] \quad r_{15} - r_{15} + 3 \\
1 & \\
2 & \quad r_{23} = r_{23} + r_3 \quad r_{24} = r_{24} + r_4 \quad r_{25} = r_{25} + r_5 \\
 & \quad \text{BR} \quad r_{13} < 30, \text{Loop} \\
0 & \quad r_2 = r_{23} + r_{24} \\
1 & \quad r_2 = r_2 + r_{25} 
\end{align*} \]
Pipelining