Contents of Lecture 12

- Instruction Scheduling Basics
- List Scheduling
- Modulo Scheduling

Instruction Scheduling Example

- The purpose of **instruction scheduling** is to improve performance by reducing the number of pipeline stalls suffered during execution.
- The following example illustrates the concept, where the right column is the scheduled code.
- Due to instructions only are scheduled within one basic block, only a limited improvement is achieved — the `fsub` and `stf` are not helped at all.

```
ldf   t2,a,t1
ldf   t3,b,t1
fadd  t4,t2,t3
ldf   t5,c,t1
ldf   t6,d,t1
fmul  t7,t5,t6
fsub  t8,t3,t7
stf   t8,e,t1
```

```
ldf   t2,a,t1
ldf   t3,b,t1
ldf   t5,c,t1
ldf   t6,d,t1
fadd  t4,t2,t3
fmul  t7,t5,t6
fsub  t8,t3,t7
stf   t8,e,t1
```

Instruction Scheduling vs. Register Allocation

- The goal of instruction scheduling is to reduce pipeline stall and this is achieved by separating the producer and consumer.
- This separation makes it more difficult to perform register allocation.
- **Question:** Which of instruction scheduling and register allocation should be performed first?
  Answer: Instruction scheduling because register allocation would create unnecessary constraints for the scheduler, and advanced instruction scheduling would be seriously limited with already assigned registers.
- If register allocation results in spill code, the instruction scheduler is usually run a second time in order to separate the load instructions from the uses of the loaded register.

Register Pressure of Different Schedules

The left schedule needs three floating point registers and the right four.

```
ldf  f2,ra,ri
ldf  f3,rb,ri
fadd f2,f2,f3
ldf  f3,rc,ri
ldf  f4,rd,ri
fmul f3,f3,f4
fsub f2,f2,f3
stf  f2,re,ri
```

```
ldf  f2,ra,ri
ldf  f3,rb,ri
ldf  f4,rc,ri
ldf  f5,rd,ri
fadd f2,f2,f3
fmul f4,f4,f5
fsub f2,f2,f4
stf  f2,re,ri
```

Rewriting Expressions for Increased Parallelism

- Consider the expression `a + b + c + d`.
- How it must be evaluated depends on the data type and source language.
- In C (and other languages) addition is left-associative which means the expression should be evaluated as `((a + b) + c) + d`.
- Due to the sequential execution it takes at least three clock cycles (and more for floating point).
- If the compiler knows that **it** can ignore the effects of overflow (either due to the type is unsigned or two's complement representation is used), it can rewrite it as `(a + b) + (c + d)`.
- On a superscalar processor the two additions can be performed concurrently.
- Programmers are **not** ignore overflow other than for unsigned integers (if the computation still makes sense with the overflow, that is).

Data Dependencies

- Data dependencies constrain how instructions can be scheduled.
- Instruction scheduling is performed after translation from SSA Form and on low level code which is close to the final machine code.
- In addition to the data dependence graph, dependencies due to scalar variables and accesses to memory through unknown addresses are used.

List Scheduling

- The most fundamental instruction scheduling technique is called **list scheduling** and schedules one basic block at a time.
- First an instruction level data dependence graph is built.
- Vertices are instructions and directed arcs constrain the scheduling.
- The source vertex must execute before the target vertex.
- List scheduling uses topological sorting.
- Once the graph has been constructed the list scheduler maintains a set of **candidate** instructions.
- Candidate instructions have no predecessor in the graph.

List Scheduler Goal

- The goal of the list scheduler is to minimize the number of clock cycles required to execute the basic block.
- As this problem is NP-complete, an approximation is found as follows: each vertex is assigned a priority in some way, and the highest priority vertex i of the candidates is scheduled next.
- Then any successor vertex s of i with no predecessor that has not yet been scheduled is moved to the set of candidates.
- This procedure is repeated until the set of candidates is empty.
- The interesting problem is to select the priority function using clever heuristics.
- Changing heuristics can change the execution time by several percent.
- In one version of the IBM C/C++/FORTRAN compiler each block was scheduled three times with different heuristics and the best schedule was used.

Storage Resources

- The instruction scheduler builds a graph based on the **definitions** and **uses** of storage resources, e.g. variables, registers, or all of memory.

Attribute	**Description**
$def(r)$	The instruction which most recently modified r while scanning backwards, or null (denoted $\perp$).
$uses(r)$	The set of instructions which use the current value of r.

Instruction s defines resource r

procedure *define_resource*(r, s)

    if $(def(r) \neq \bot)$ {
        add edge $(s, def(r), \mathrm{OUTPUT})$
        delete $def(r)$ from *candidates*
    }

    $def(r) \leftarrow s$

    for each $u \in uses(r)$ **do** {
        add edge (s, u, TRUE)
        delete u from *candidates*
    }
    $uses(r) \leftarrow \emptyset$

end

Instruction s uses resource r

procedure *use_resource*(r, s)
    add s to *uses*(r)
    if (*def*$(r) \neq \perp$ **and** *def*$(r) \neq s$)
        delete *def*(r) from *candidates*
        add edge (s, *def*(r), ANTI)
end

Collecting Candidate Instructions

```
procedure collect_candidates(v)

    /* v is a basic block. */

    for each resource r do {
        uses(r) ← ∅
        def(r) ← ⊥
    }
    candidates ← ∅
    for each instruction s in v in reverse order do {
        for each resource r defined by s do
            define_resource(r, s)
        for each resource r used by s do
            use_resource(r, s)
        add s to candidates
    }
end
```

List Scheduling

```
procedure list_sched
        for each vertex v in G do
                collect_candidates (v)
                cycle ← 0
                while (candidates ≠ ∅) do
                        for each s ∈ candidates do
                                update_earliest (s)
                                compute_delay (s)
                        max_delay_cand ← ∅
                        earliest_cand ← ∅
                        for each s ∈ candidates do
                                if (delay (s) = max_delay)
                                        add S to max_delay_cand
                                        if (earliest (s) ≤ cycle)
                                                add S to earliest_cand
                        if (earliest_cand ≠ ∅)
                                take s from earliest_cand using heuristics
                        else
                                take s from max_delay_cand using heuristics
                        delete s from candidates
                        schedule s as next statement in v
                        cycle ← cycle + 1
end
```

- Incrementing the cycle after each scheduled instruction assumes a single-issue pipeline.

Modulo Scheduling

- Consider the following loop and assume there are true dependencies from A to B and from B to C.

```
void h()
{
        int     i;

        for (i = 0; i < 100; ++i) {
                A;
                B;
                C;
        }
}
```

- Due to list scheduling only works with one basic block, it cannot improve this loop.
- Such loops are of course quite common.

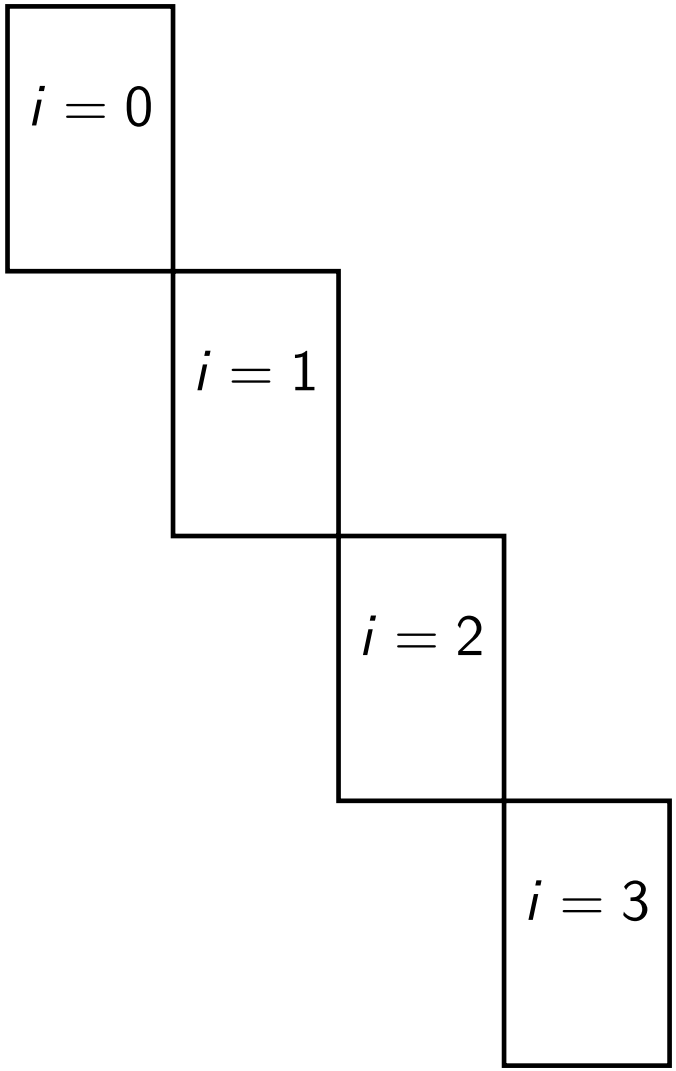
Modulo Scheduling the Loop

- Let us take instructions from three iterations and interleave them.
- First we need to execute instructions from the first two iterations in a prologue.

cycle	i	ii	iii
0	A_0		
1	B_0	A_1	
2	C_0	B_1	A_2
3	A_3	C_1	B_2
4	B_3	A_4	C_2
5	C_3	B_4	A_5
6	A_6	C_4	B_5
7	B_6	A_7	C_5
8	C_6	B_7	
9		C_7	

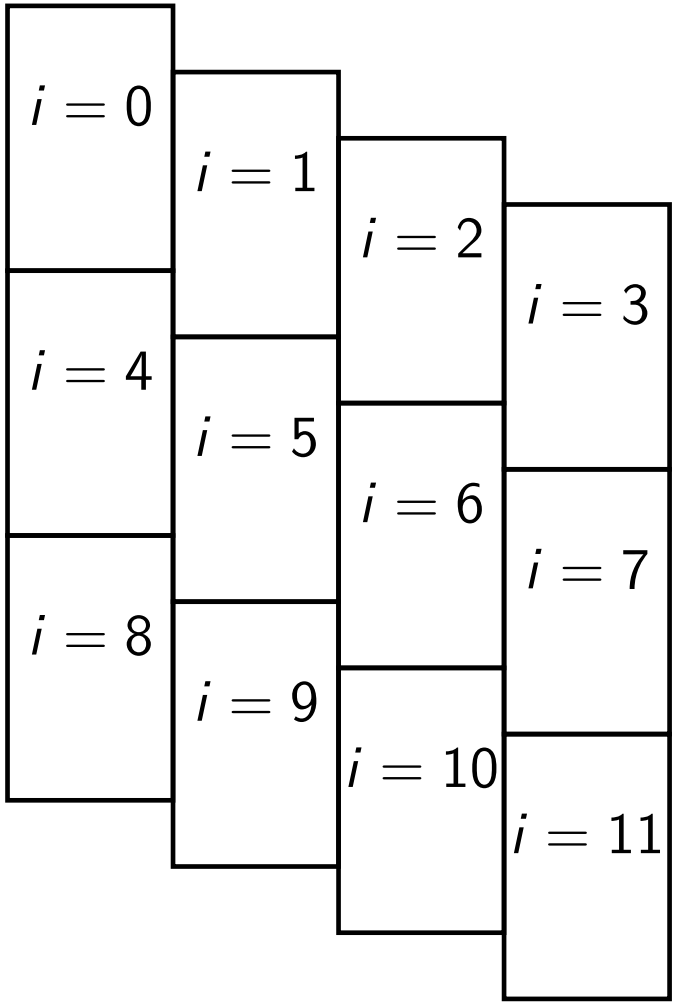
- Assume for illustration only 8 iterations are executed.
- For example A_3 denotes instruction A in iteration 3.
- After a steady-state with 2×3 iterations there is an epilogue.
- Consider instruction B_3. While it waits for A_3, the CPU can also execute C_1 and B_2, assuming a pipelined superscalar CPU.

List Scheduled Execution



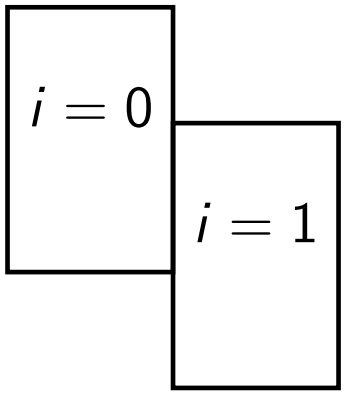
- Each iteration is completed before the next starts.
- The height of an iteration is the number of clock cycles it takes.

Parallelism with Modulo Scheduling



- A new iteration is started before the current has completed.
- We wish to start the next iteration as early as possible.
- If we start the next iteration the **same** clock cycle, we need a multicore with one core per loop iteration.

The Initiation Interval



- The **initiation interval**, abbreviated **II**, is the number of clock cycles between the start of two iterations.
- The II is limited by
  1. Data dependencies
  2. Available hardware resources
- A maximum II is determined by doing a normal list schedule of the loop body.
- A minimum II is computed from the available resources and required resources in the loop, and the data dependencies.

Performing Modulo Scheduling

- The modulo scheduler then tries to find the smallest II which results in a valid schedule, by trying each value of II starting from the minimum II, and incrementing it by one.
- All variables defined before being used in each loop iterations are expanded to different variables for each iteration.
- Then the loop body is duplicated and adapted for the proper iteration.
- A prologue and an epilogue is also generated.

Data Dependence Analysis for Modulo Scheduling

- There are two data dependence analyzes done for modulo scheduling:
  1. Instruction level — as we saw for list scheduling.
  2. Loop level — as we saw in lecture F10.
- There is one modification: in addition to the type (true, anti, or output), dependencies for modulo scheduling are of the form (p, d), where p is the dependence distance (i.e. iteration difference) and d is the delay in clock cycles.
- By delay is meant the time the instructions should be separated to avoid pipeline stalls.

Scheduling Instructions

- Let $\sigma(v)$ denote the clock cycle a certain instruction v is scheduled.
- With a dependence (p, d) from instruction u to instruction v, and an initiation interval II, to avoid pipeline stalls we need to satisfy:

$$\sigma(v) - \sigma(u) + p\ II \geq d \tag{1}$$

- Additionally there must be sufficient hardware resources available in each clock cycle.
- Assume for simplicity an instruction only needs one resource each clock cycle (e.g. a certain stage in a pipelined functional unit).
- Then for each clock cycle i the instruction executes (counting from zero in e.g. in the instruction decode stage) there must be such a resource available in the clock cycle given by:

$$(\sigma(v) + i) \mod II \tag{2}$$

- If no value for $\sigma(v)$ can be found which satisfies all constraints, the initiation interval must be increased, and the scheduling be repeated.

Modulo Scheduling Algorithm

- We have now seen the essential parts of the modulo scheduling.
- There was in the 1980's a debate regarding which hardware features were needed for efficient software pipelining (e.g. rotating register files).
- The problem was solved completely in software by Monica Lam in her PhD thesis from Carnegie Mellon University.
- Her algorithm is described in her book "A Systolic Array Optimizing Compiler", and is implemented in several compilers, including in SGI's compiler (now called Open64).
- An interesting study was performed that compared an optimal scheduler with the modulo scheduler in the SGI compiler, and concluded that their modulo scheduler almost always produced optimal code.