Static Code Scheduling for Parallelism

Jack Romo

Prerequisites

- **Implementations of Programming Languages**, Jeremy Jacobs
- **Introduction to Computer Architecture**, Mike Freeman
- *(Desirable)* **Systems Part 1**, Neil Audsley
- A knowledge of the following concepts from the above:
    - Compilers
        - Register Allocation
        - Control Flow Graphs, Liveness
    - Computer Architecture
        - Pipelines
        - Caching
        - Out-Of-Order Execution

Course Overview

- A focus on the optimization phase, more exactly on hardware-level parallelism
  - We assume a ‘reasonably efficient’ total map from source to target language
- We will not cover issues with optimization-cognizant code generation
- An overview of optimizing for instruction-level parallelism, including:
  - Data and Control Dependency
  - Basic-Block and Global Scheduling
  - Software Pipelining
- This course follows chapters 10 and 11 of ‘Compilers: Principles, Techniques and Tools’, Alfred V. Aho et al.

Instruction-Level Parallelism

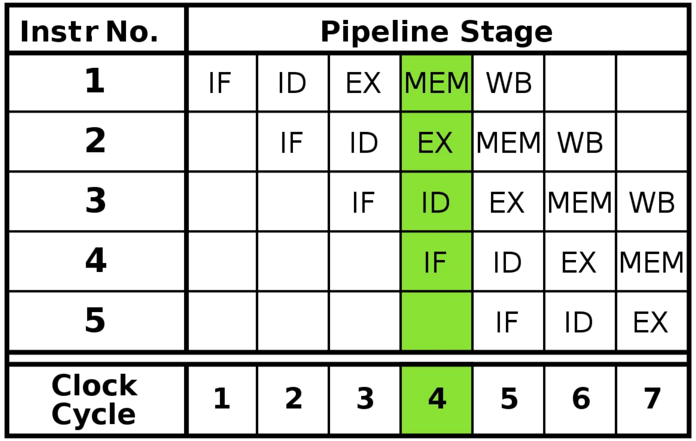
Introduction to Scheduling

Lecture 1

- Hardware Parallelism
- Data Dependence
- Basic Block Scheduling

Hardware Parallelism: The Pipeline

- Our code is assumed to run on a standard instruction pipeline
- Pipelines are fine until we perform a:
  - Jump instruction
  - Memory load
  - Memory store
- General-purpose processors can detect dependences between instructions
  - Basic schedulers will wait until current instruction’s data dependencies resolved
  - More complex one might perform ‘out-of-order execution’ while waiting for data



Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Static Scheduling

- Even if hardware does scheduling, it can’t schedule what it hasn’t seen!
- Compiler may do **static scheduling** beforehand to help
- Can help by:
    - Ensuring instructions at start of a branch are parallelizable
    - Placing independent instructions nearby to one another

Code Scheduling Constraints

- Control Dependence Constraints
  - All operations executed in original program must be executed in the optimized one
- Data Dependence Constraints
  - Operations in the optimized program must produce same results as in original program
- Resource Constraints
  - Schedule must not oversubscribe resources on the machine

Data Dependence

- Alongside control dependence, the “main challenge” of scheduling
- 3 types of data dependence between instructions:
  - **True Dependence**: read after write
    - An instruction depends on result of another
  - **Antidependence**: write after read
    - An instruction will overwrite what another needs
  - **Output Dependence**: write after write
    - Memory location must be left in a certain state
- Antidependence and output dependence are **storage dependencies**, not true dependencies on results of operations per se
  - Can be rectified simply by writing to another location instead
  - True dependence is more ‘fundamental’ (cause and effect vs. storage constraints)

Dependence Across Memory Accesses

- Take a C-like language with pointers
- Cannot assume 2 pointers point to different locations
    - To deduce if they do requires running the code, **undecidable** and dependent on OS state
- If cannot be disproven, assume references are to same memory location
    - This is not an issue in type-safe languages!
- Many different forms of dependence analysis:
    - Array Data-Dependence Analysis
    - Pointer-Alias Analysis
    - Interprocedural Analysis *(not covered here!)*

```
1)  a = 1;
2)  *p = 2;
3)  x = a;
```

What are the dependencies?

Register Allocation vs. Scheduling

- Consider the code to the right
- If t1 and t2 are allocated to separate registers, have 2 data dependencies
  - True dependency from (1) to (2)
  - True dependency from (3) to (4)
- If t1 and t2 are allocated the same register, we gain more dependencies
  - Output dependency from (1) to (3)
  - Antidependency from (2) to (3)
- More constraints on scheduling

```
1)  LD t1, a    // t1 = a
2)  ST b, t1    // b = t1
3)  LD t2, c    // t2 = c
4)  ST d, t2    // d = t2
```

Solution?

- Use a scheduling-cognizant allocation algorithm
    - Not trivial
    - Balance between optimizing for parallelism and for register usage remains
    - ‘Best’ solution in the end
- Use **hardware register renaming**
    - Allow compiler to generate highly register-usage-optimized code with many dependencies
    - Hardware gives many registers the same name
        - Runs instructions with register allocation incurred dependencies to run in parallel, accessing different aliases to same register
    - Common for early compilers, shows the effect compiler design can have on architecture

A Basic Machine Model

- Represent a machine as M = <R, T>
- T = set of all operation types (load, store, add, jump, ...)
- R = [r_1, r_2, ...] = array of resources available
    - r_i = number of resources available of type i
    - Includes ALU's, memory access units, FPU's, etc.

Operation Requirements

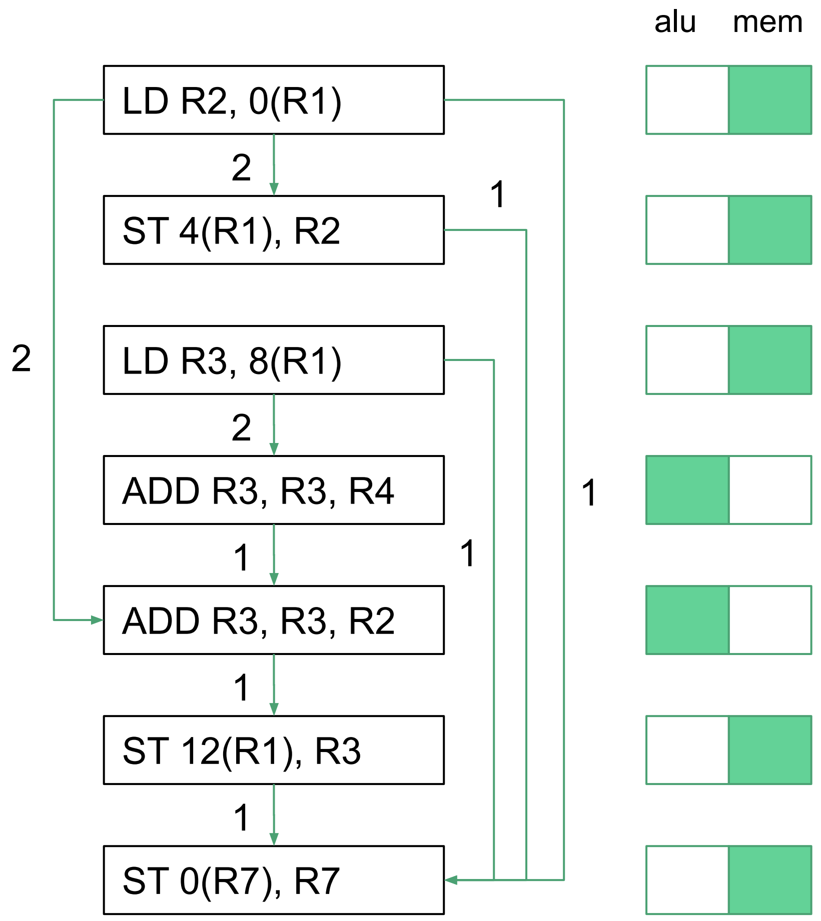
- Each operation has set of input / output operands and a resource requirement
  - Each input operand has a latency, giving when the input is needed by after the instruction starts, in clock cycles
    - Typically = 0, ie. needed immediately
  - Output operands have latency telling when the operand is finished with after the instruction starts, in clock cycles
- Resources have times for when they are needed by instruction after it starts
  - Given by a **resource reservation table**, RT_t
  - $RT_t[i, j]$ = number of units of jth resource operation t needs i clock cycles after commencing
- Operation is **fully pipelined** if it can be executed at every clock
  - Don’t need to model every stage of pipeline, just one for first stage will do
  - Op in first stage of pipeline is guaranteed right to proceed to later stages as clock proceeds

Basic Block Scheduling

- We can now start considering a scheduling algorithm for basic blocks
  - A basic block is a sequence of code where every operation in the sequence is guaranteed to run if the first is guaranteed to do so
- Doing this optimally is NP-complete!
  - Basic blocks tend to have many data dependencies / are small, so not an big issue
- We will introduce an algorithm called **list scheduling**
  - Simple but ‘good enough’

Data-Dependence Graphs

- We represent a basic block of instructions with a **data-dependence graph**, where G = (N, E)
  - N = nodes, each is an instruction
  - E = edges, an edge from *x* to *y* with value *n* means *y* must start no earlier than *n* clock cycles before *x* starts; indicates a data dependence between instructions
- Consider the graph to the right
  - Machine can do 2 instructions per clock
  - One must be a branch or ALU op
  - The other must be a load or store, load takes 2 clock cycles and is fully pipelined



List Scheduling

- **Input:** A machine M = <R, T> and a data-dependence graph G = (N, E).
- **Output:** A schedule S that maps operations in N to time slots in which those operations can be initiated, satisfying all data and resource constraints.
- **Method:**
  - Start with an empty reservation table RT and an empty map S.
  - For each *n* in N in a **prioritized topological order**:
    - Let $s = \max_{e=p\to n \text{ in } E}(S(p) + d_e)$ *(d_e = weight of edge 'e')*
    - While there exists an *i* such that $RT[s + i] + RT_n[i] > R$, **do** s += 1
    - $S(n) = s$
    - For all *i*, **do** $RT[s + i]$ += $RT_n[i]$
    - Repeat for the next *n* in N. If have gone through all nodes, return S.

Prioritized Topological Order

- A heuristic priority function for nodes to be scheduled (no backtracking)
- Possible prioritized orderings:
    - **Critical path:** without resource constraints, shortest schedule given by longest path through data-dependence graph
        - Good metric = **height** of a node, the longest path that starts with said node
    - **Resources:** If all operations independent, length constrained by this
        - Operations using **more critical resources** given priority
    - Can finally use source ordering to break 'ties', where op that is first in source goes first

Example Result

- In previous example, critical path (including time to execute last instruction) = 6 clock cycles (load of R3 to store of R7)
- Use height as priority function

Schedule

	LD R3, 8(R1)
	LD R2 0(R1)
ADD R3, R3, R4	
ADD R3, R3, R2	ST 4(R1), R2
	ST 12(R1), R3
	ST 0(R7), R7

alu	mem
	🟩
	🟩
🟩	
🟩	🟩
	🟩
	🟩

Global Scheduling

Lecture 2

- Control Dependence
- Speculative Execution
- Global Scheduling
- Region-Based Scheduling

Control Dependence

- Scheduling within basic block easy; all ops guaranteed to execute
- However, basic blocks are generally small and have high data dependence
    - Thus, must exploit parallelism **across** basic blocks
- An operation is **control dependent** on another if whether it executes is dependent on the latter’s result
    - Body of a while loop is control dependent on the predicate

Speculative Execution

- Sometimes, we can **speculatively execute** instructions at a branch
  - Pipeline will allow an instruction to start executing as jump is processed
  - If we can avoid unwanted side effects, we gain speed if we take the branch with said instruction
- Unwanted side effects: memory accesses
  - Speculatively executing a memory access may result in an invalid access
  - This could cause errors or page faults (extremely costly!)

```
int *p;
// …code here...
if(p != NULL) {
    q = *p;
}
```

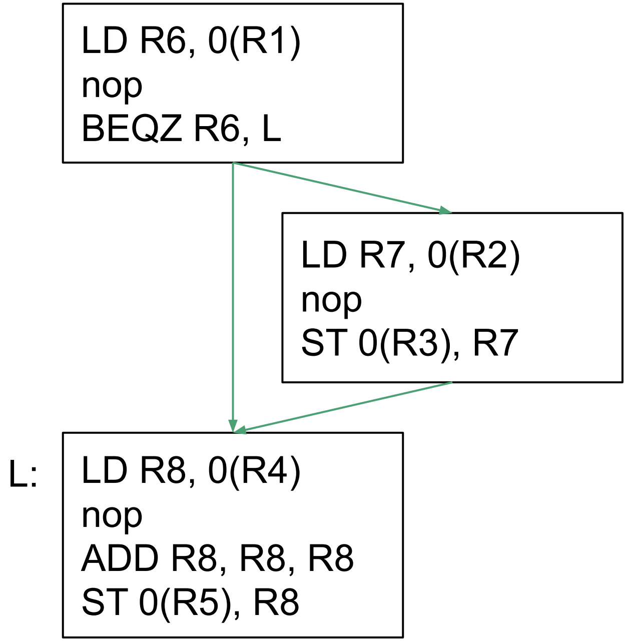
Why can’t we speculatively execute?

Hardware Support for Speculative Execution

- Prefetching
  - Special ‘prefetch’ instruction, indicates to processor some memory is likely to be used
  - If invalid or causes a page fault, processor ignores
  - Otherwise brings it into cache if not already there
- Poison Bits
  - Registers each have a special ‘poison bit’ flag
  - If invalid speculative load into said register occurs, no exception, poison bit is just set to 1
  - If register with poison bit = 1 accessed, this is when exception occurs
- Predicated Execution
  - Adds a special instruction that only executes on a predicate, alternative to a jump
  - Converts control dependence into data dependence
  - Predicated instructions add data dependence / need resources, shouldn’t be overused

Global Code Scheduling

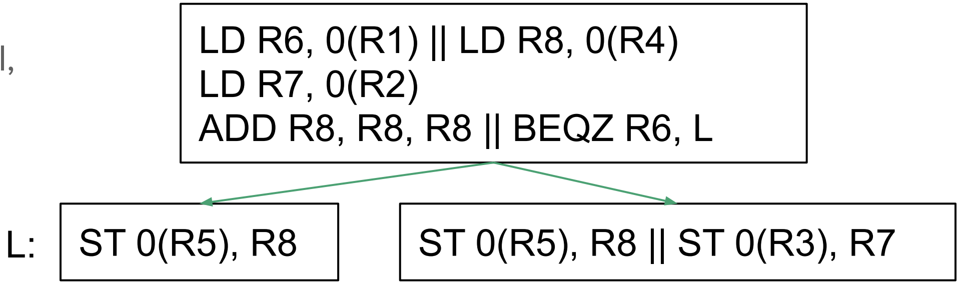
- To leverage speculative execution and bypass high in-block data dependence, we must consider movement of code **across basic blocks**
- Consider the example to the right:
    - Loads take 2 clock cycles, everything else takes 1
    - Can run any two instructions in parallel, we denote parallel instructions with ‘||’



```
LD R6, 0(R1)
nop
BEQZ R6, L
```

```
LD R7, 0(R2)
nop
ST 0(R3), R7
```

```
L:  LD R8, 0(R4)
    nop
    ADD R8, R8, R8
    ST 0(R5), R8
```



```
LD R6, 0(R1) || LD R8, 0(R4)
LD R7, 0(R2)
ADD R8, R8, R8 || BEQZ R6, L
```

```
L:  ST 0(R5), R8
```

```
ST 0(R5), R8 || ST 0(R3), R7
```

Block dominance

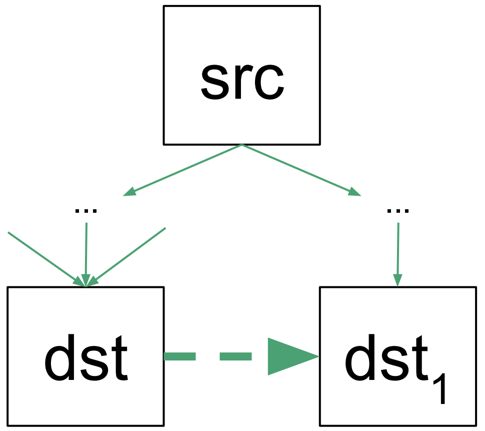
- We must consider what control paths are possible through basic blocks
  - Achieved with a **control flow graph** (see IMPL)
  - We may move code to an earlier block only if the first is the only path to the latter, and vice versa; we require a formalization of this
- Let there exist a control path through first block A, then block B
  - We say block A **dominates** block B if every control path through B also passes through A
  - We say block B **postdominates** block A if every control path through A also passes through B
- If block A dominates block B and block B postdominates block A, we say they are **control equivalent**
  - In this case, block A is executed if and only if block B is executed
  - Code can be moved between control equivalent blocks freely; no extra / lost instructions
- We must therefore consider all other cases

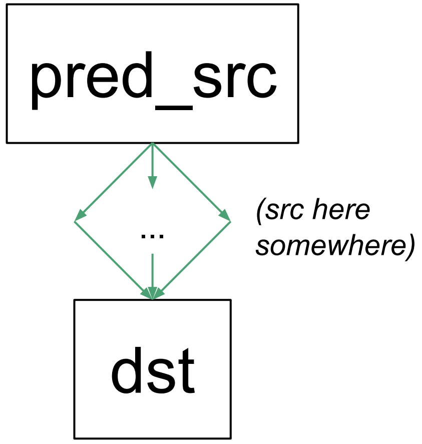
Upward Code Motion

- Say our source block *src* is in a control path after our destination block *dst*
- If *src* **does not postdominate** *dst*:
  - Then there exists a path through *dst* that does not go through *src* later
  - If code moved only uses otherwise idle resources, will be beneficial if and only if *src* is chosen to run (ie. **speculative execution**)
    - Will be illegal if the moved code affects resources used in different paths through *dst*
- If *dst* **does not dominate** *src*:
  - Then there exists a path through *src* that does not go through *dst* first
  - Must move code to all paths that go to *src* (called a **cut set** of blocks, which separate entry from *src*)
  - If operands will have same value, instruction result doesn't overwrite a needed value, and the instruction result isn't overwritten before reaching *src*, this **compensation code** is allowed
    - This may make some paths slower; optimized paths must run more frequently

Downward Code Motion

- Say our source block *src* is in a control path before our destination block *dst*
- If *src* **does not dominate** *dst*:
    - Then there is a path through *dst* that avoids *src*
    - If we move a write op to *dst*, may overwrite values in a path that avoids *src*
    - Can remove *src*-to-*dst* path, make a duplicate src-to-*dst*$_1$ path
        - This makes *src* dominate the new *dst*$_1$, can move ops here
    - Can also change instruction moved to *dst* to be **predicated**, such that it only executes if went through *src* initially
        - Must have *dst* be dominated by block that provides predicate, may have a path where predicate is unavailable otherwise



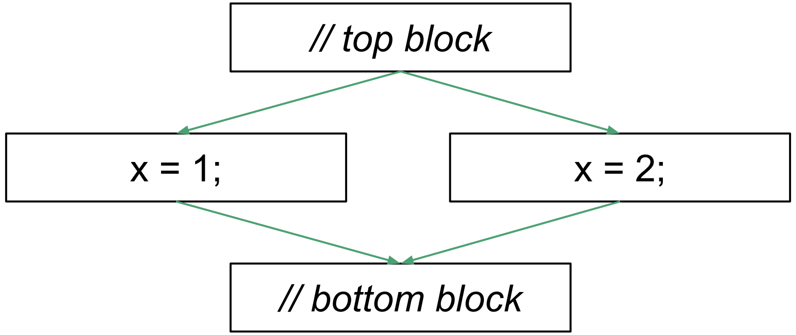


Downward Code Motion (continued)

- If *dst* **does not postdominate** *src*:
  - Need compensation code on all paths not going to *dst*, ie. on the **cut set** that separates *src* from the exit

Updating Data Dependences

- We must update data dependences upon each code motion
- Observe example to the right
    - 3 basic blocks
    - *x* is not live on exit of top block currently
    - However, it will be live on exit if we move either assignment to *x* up
    - This means we can't move both assignments up, **only one**
        - Must update dependences before attempting second movement to see this



Considerations when scheduling code

- Not all instructions are created equal!
  - About 90% of execution time is on average spent on ~10% of the code
- Should thus aim to make frequently executed paths run faster generally
- Can use estimates such as:
  - Instructions in inner loops execute more frequently than those in outer loops
  - Branches that go backward are more often taken than not
  - Branches that can go to an exit are unlikely to execute
- May also monitor code as it is run, then feedback information to compiler
  - This is called **dynamic profiling**

Region-Based Scheduling

- We now introduce an algorithm that covers the two easiest code motions
  - Move operations up to control-equivalent basic blocks
  - Move operations speculatively up one branch to dominating predecessor
- Define a **region** to be a subset of control-flow graph accessible via only one entry block; may represent any program as a hierarchy of these
  - We will go through all regions, innermost first; we ignore code in regions we aren’t currently in
  - Once we reach one, we iterate through all blocks B
    - We collect all blocks control equivalent to B and dominated by blocks control equivalent to B and take their instructions
    - If we can do an instruction from any of these while in B, do them in B
    - We will iterate through all blocks in such an order that doing this puts instructions in ‘the best place’ (no change if control equivalent, speculative performance gain if dominant)

Region-Based Scheduling Algorithm

- **Input:** A control-flow graph and machine-resource description.
- **Output:** A schedule S mapping instructions to a block and time in said block.
- **Method:**

```
○ For each region R in topological order, so inner ones go first:
    ■ Compute data dependences
    ■ For each basic block B in R in prioritized topological order:
        ● CandidateBlocks = ControlEquiv(B) U DominatedSuccessors(ControlEquiv(B))
        ● CandidateInstructions = ready instructions in CandidateBlocks
        ● For t = 0, 1, …, until all instructions in B are scheduled:
            ○ For each instruction n in CandidateInstructions, in priority order:
                ■ If n has no resource conflicts at time t:
                    ● S(n) = <B, t>, update resource commitments / data dependences
                ■ Update CandidateInstructions
```

Notes on Region-Based Scheduling

- Topological order of blocks in a region:
  - Control flow and data dependence graph edges going back to entry block of region ignored
    - This makes the graph acyclic
  - Blocks are not scheduled until all instructions they depend on have been scheduled
  - Will begin at entry block to region
- Topological order of instructions:
  - Same as in list scheduling, but instructions in a control equivalent block are higher priority than in a dominated successor only
    - Latter type of instructions only speculatively executed, could lower performance

Loop Unrolling

- Loop body forms a region, means we cannot move code between iterations
- Solution: 'unroll' the loop body several times to bypass this
- The change from the top example to bottom shows this; 3 unrolls done here
    - More unrolling permits algorithm to schedule for better parallelism between iterations

```
for(i = 0; i < N; i++)
    S(i);
```

```
for(i = 0; i+3 < N; i+=3) {
    S(i);
    S(i+1);
    S(i+2);
}

for(; i < N; i++)
    S(i);    // finish up
```

Neighborhood Compaction

- The given algorithm does not support compensation code, even if it would be beneficial
- We may add this by doing a pass over every pair of blocks, where one block runs directly after the other
    - If code can be moved up/down between and is beneficial, find other paths to/from these blocks and insert compensation code into them too if net gain is profitable
- This is helpful in loops for example; can move operations from end of loop to start and from start to outside of loop
- However, only works for adjacent blocks, so impact is limited

Interacting with Dynamic Schedulers

- If the given system has a dynamic scheduler, the compiler can do many things to help it achieve optimal performance
    - Can add prefetch instructions early on if available, to avoid cache misses
    - If not, can move instructions with likely cache misses earlier instead
    - Static scheduler can simply put instructions ‘in order’, dynamic scheduler can resolve data dependencies on the fly
- If not, must be more conservative
    - Must separate instructions by their data dependence
        - Should assign high delay to likely dependences, short otherwise
    - Must ensure instructions along uncommon branch paths are efficient; branch misprediction is costly

Software Pipelining

Lecture 3

- Do-All and Do-Across Loops
- Software Pipelining
- Scheduling Acyclic and Cyclic Dependence Graphs

The Challenge

- In numerical applications, massively parallelizable ‘do-all’ loops are commonplace
- It is up to the compiler to ensure machine resources are saturated
- Optimizing loops is only really beneficial when done ‘across’ iterations
    - Loop unrolling helps somewhat; however, it results in larger code
    - We can do better!

Our Example Machine

- The machine can issue one load, one store, one arithmetic operation, one branch operation
- Has a loop-back instruction for convenience: **BL R, L**
  - Decrements *R* and branches to *L* if the result is 0
- **(R++)** insertable in register usage to increment *R*'s value during the instruction
  - The effect of the increment is not visible until after the instruction runs
- Arithmetic operations are **fully pipelined**, ie. they can be initiated every clock
  - Their results, however, do not become visible until 2 clocks after they start
- All other operations have single-clock latency

Our Running Example

- Consider the following loop and its compiled counterpart:

```
for(i = 0; i < n; i++)
    D[i] = A[i]*B[i] + c;
```

```
// R1, R2, R3 = &A, &B, &D
// R4 = c
// R10 = n - 1

L:    LD R5, 0(R1++)
      LD R6, 0(R2++)
      MUL R7, R5, R6
      NOP
      ADD R8, R7, R4
      NOP
      ST 0(R3++), R8  ||  BL R10, L
```

Loop Unrolling Inadequacies

- We can get better performance via loop unrolling, as we have seen
- However, unrolling more means more code
  - This may be undesirable
- We show the previous example unrolled 4 times here
  - Register usage is ignored
- Looks like a software-level ‘pipeline’, where the pipeline gets flushed every 4 instructions
  - Can we stop this periodic flushing?

```
L:  LD
    LD
         LD
    ML   LD
         ML   LD
    AD        LD
         AD        LD
    ST        ML   LD
         ST        ML
              AD
                   AD
              ST
                   ST   BL (L)
```

The Insight

- Observe the unoptimized loop done 5 times here, with overlap where possible
    - Register allocation ignored again
- Lines **7-8** are **exactly the same** as lines **9-10**
- We can iterate by just repeating the middle bit!
    - It is a ‘pipeline’ of 5 parallel iterations, doing instructions 1-2 of the first, instructions 3-4 of the next, etc.

1)	LD				
2)	LD				
3)	ML	LD			
4)		LD			
5)		ML	LD		
6)	AD		LD		
7)			**ML**	**LD**	
8)	**ST**	**AD**		**LD**	
9)				**ML**	**LD**
10)		**ST**	**AD**		**LD**
11)					ML
12)			ST	AD	
13)					
14)				ST	AD
15)					
16)					ST

Software Pipelining

- Our optimized code is thus shown here (ignoring registers again)
- Lines 1-6 is called the **prolog**
    - Filling the ‘pipeline’
- Lines 7-8 is called the **steady state**
    - The ‘pipeline’ is at full efficiency
    - 4 iterations being done at once
- Lines 9-14 is called the **epilog**
    - Emptying the ‘pipeline’
- This only works if more than 3 iterations are performed!
    - Can do < 4 iterations separately

```
 1)         LD
 2)         LD
 3)         ML   LD
 4)              LD
 5)              ML   LD
 6)         AD        LD
 7)   L:              ML   LD
 8)         ST   AD        LD   BL (L)
 9)                        ML
10)              ST   AD
11)
12)                   ST   AD
13)
14)                        ST
```

Considering Register Allocation

- Consider our newly pipelined example
    - Two multiplications are produced by overlapping iterations, used by two addition operations later
    - This means we need 2 registers for the 2 results of the multiply operation!
- We will have odd-numbered and even-numbered iterations use different sets of registers; this turns out to solve all our conflicts!
    - We will formalize this notion later

Goals and Constraints of Software Pipelining

- Given a data-dependence graph G = (N, E) for our loop, we can specify our schedule by:
  - An **initiation interval** T (number of cycles from one iteration starting to the next)
  - A relative **schedule** S (schedule for one iteration, such that it can be pipelined with T)
- An operation in the *i*th iteration (counting from 0) is executed at clock Ti + S(n)
- We must consider:
  - Resource constraints across parallel loops
  - Data dependences, both within an iteration and across many
    - This leads to cyclic data dependence graphs!

Modular Resource Reservation

- We wish to find resource usage in the steady state
  - The prolog and epilog will necessarily work if the steady state does
- NB: If an iteration requires n_i units of resource *i*, the initiation interval T must be at least $max_i(n_i/r_i)$. (Think about this!)
- We use a **modular resource reservation table** to model resource usage in the steady state, called RT_s
- It takes the form:

$$RT_s[i] = \Sigma_{\{t \mid t \bmod T = i\}} RT[i]$$

Data-Dependence and Initiation Intervals

- In loops, we may have cyclic data dependence graphs
    - We thus gain an upper and lower bound on where instructions can be scheduled relative to one another
- We augment our data-dependence graph such that each edge has two values *<f, d>*, where instruction 1 occurs *d* clocks and *f* iterations before instruction 2
- We gain another constraint on T in a data-dependence graph G = (N, E):

$$T \geq \max_{c \text{ a cycle in } G} \left[\left(\Sigma_{e \in c}\, d_e \right) / \left(\Sigma_{e \in c}\, f_e \right) \right]$$

Data-Dependence and Scheduling

- Suppose we have two instructions *n* and *m* in a cycle
- We thus have a path $n \rightarrow m$ with accumulated edge values $<f_1, d_1>$ and another path $m \rightarrow n$ with accumulated values $<f_2, d_2>$
- We gain two constraints, which give us upper/lower bounds on S(*m*):

$$f_1T + S(m) - S(n) \geq d_1$$

$$f_2T + S(m) - S(n) \geq d_2$$

$$\Rightarrow \; S(n) + d_1 - f_1T \quad \leq \quad S(m) \quad \leq \quad S(n) - d_2 + f_2T$$

SCCs and Simple Graphs

- A **strongly connected component** (SCC) is a subset of a graph such that all nodes can reach one another by some sequence of connected nodes
- Given *n* and *m* are on a path *p*, we have that

$$S(n) - S(m) \quad \geq \quad \Sigma_{e \in p}(d_e - f_e T)$$

- However, if *p* is an SCC (ie. a cycle), we have that:
    - Around *p*, the f_e's must add up to a nonnegative value (if ≤ 0, then an instruction in *p* would precede itself or has to be executed at the same clock for all iterations)
    - The sum of the delays is ≤ T (think about *p* in the steady state)
- Thus, the above RHS must be ≤ 0
    - This means cycles give us very little constraint on S(*n*) - S(*m*)
    - We should find non-cycle paths, called **simple paths**, for this

Generating a Schedule

- We will not cover the exact algorithm here (see pg. 755)
- We iterate through several possible initiation intervals, with a starting point such that the previous constraints on T are satisfied
- For each interval, we try and schedule every SCC one at a time in the data-dependence graph; if impossible, we try the next T
    - From before, we have ranges that each instruction can be scheduled in by data dependences
    - We start with an empty modular reservation table *RT*
    - We try every possible schedule for instructions in the above ranges; if *RT* is oversaturated by a schedule, we backtrack and try another, if no configuration works then try another T
- Scheduling this is an NP-complete problem
    - This is why we use a ‘guess-and-check’ backtracking algorithm

Modular Variable Expansion

- We sometimes have variables that have a lifetime within one iteration
- We call these variables **privatized**
- When we have these, we may do **modular variable expansion** - we turn the variable into an array of variables, where each iteration uses one entry in the array
- If a privatized variable has lifetime *l*, we only need *q* = *l*/T instances in the array at any one time
- We will, however, need several versions of the same iteration, to use each set of registers

Conditional statements

- If conditionals exist in the loop, we may:
  - Use predicated instructions, if they are available
  - Consider only the union / sum of all the dependencies of every path; we will be able to do our pipelining as per normal then, regardless of the control dependences
    - This requires more machinery, not discussed here!