LSU EE 4720 # Dynamic Scheduling Study Guide **Fall 2005**

David M. Koppelman

1.1 Introduction

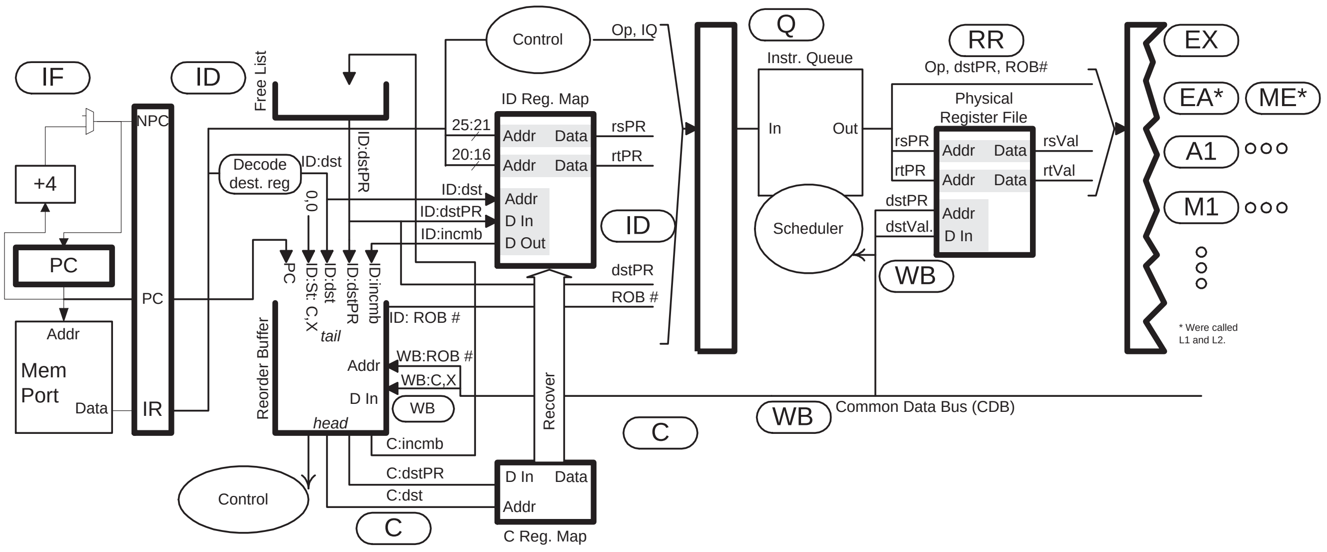
The material on dynamic scheduling is not covered in detail in the text, which is unfortunate since as of this writing most general-purpose processors are dynamically scheduled. This study guide provides a summary of dynamic scheduling and a guide to sample problems from old homeworks and exams. (The solutions are sometimes detailed and so are a valuable study resource.)

In a *statically scheduled processor* instructions start execution in program order while in a *dynamically scheduled processor* instructions can start execution out of order. Statically scheduled systems are much simpler since instructions march through the pipeline in step, with bubbles (gaps) inserted where necessary. A shortcoming is that an instruction that must wait, say for an operand, blocks all of the instructions ahead of it (towards IF). Those instructions with dependencies on the waiting instruction would have to wait anyway but there is no reason to block other instructions (other than hardware cost). In a dynamically scheduled system the only instructions that normally must wait are those for which input operands are not ready. (There are other reasons for waiting, for example, the needed functional unit is busy.) Dynamically scheduled processors are far more costly but achieve better performance than static scheduling on superscalar processors, especially when loads miss the cache (covered later) and so take more than a cycle or two.

As of this writing most general-purpose processors are dynamically scheduled. This includes the Pentium III and Pentium 4, Alpha 21264, MIPS R10000, PowerPC 620, and HP-PA 8000, and the later versions of these processors. Two exceptions are the Sun UltraSparc III and the Intel Itanium 2, which are statically scheduled.

1.2 Summary of Dynamic Scheduling Method 3

In class three methods of dynamic scheduling were mentioned, but (in Fall 2003) only one was covered in detail, Method 3. In Method 3 register values are stored in a *physical register file* and *physical register numbers* are used to re-name registers. This is the only method covered in the Fall 2003 semester and so Fall 2003 final exam questions are unlikely to use the other methods. The following is a brief description of Method 3. First, activities in each stage are covered, then the tables and other elements are described. The descriptions below refer to the following illustration:



1.2.1 Stage: `IF`, Instruction Fetch

This occurs strictly in program order. The number of instructions that can be fetched per cycle is equal to the *decode width*. (An x-way superscalar processor has a decode width of x.)

For simplicity `IF` is shown using the same hardware as the one-way (scalar) statically scheduled processor. In real dynamically scheduled superscalar systems additional hardware is needed for branch and target prediction and for shifting and masking instructions retrieved from the memory port (really a cache port).

1.2.2 Stage: `ID`, Instruction Decode

This occurs strictly in program order. The maximum number of instructions that can be decoded per cycle is equal to the decode width. The following is done:

An entry for the instruction is placed in the reorder buffer with the following information: The address of the instruction (`PC`); two status bits: `C` (complete) and `X` (raised an exception); the architected destination register number, `dst`, (*e.g.*, `t1`, `s2`); the physical destination register, `dstPR`, (*e.g.*, `p90`, `p103`); and the incumbent physical register number, `incumb`.

The *incumbent* is the physical register holding the **previous** value of the destination register (*not* the value that the instruction will write). For example, consider the `sub` instruction in the example below. The physical registers used for the destinations are shown in the comments. When `sub` is in ID physical register 92 is removed from the free list and is used for `t1`, the architected register holding the result of the subtract. The incumbent or "old" value of `t1` is the value written by the `add` which is in physical register 90.

```
add t1, t2, t3  # t1 to be stored in physical register 90
or  t5, t1, t7  # t5 to be stored in physical register 91
sub t1, t8, t5  # t1 to be stored in physical register 92
```

Also in `ID`, the architected source register numbers are translated into physical registers using the ID register map. The source physical register numbers are labeled `rsPR` and `rtPR` in the diagram.

A physical register is allocated by removing it from the free list and assigning it to the destination. This new physical register is labeled `dstPR` on the diagram. For `sub` in the example above `dstPR` would be 92.

The ID map is updated using this new physical register. First the incumbent register, 90, is read from the ID map (using the architected destination, `t1`, as the address) and placed in the instruction's ROB entry. (If `sub` commits the incumbent is put back in the free list.) Then the new physical register, 92, is written into the ID map, overwriting the incumbent.

The control logic determines which instruction queue to put the instruction in (`IQ` in the diagram) and which operation to perform (`Op`). (The diagram only shows one instruction queue.) The reorder buffer provides a unique number associated with the new ROB entry, `ROB #`, this is used during `WB`.

1.2.3 Stages: `Q`, Instruction Queue; `RR`, Register Read

Fall 2003 and later: In the `Q` stage the instruction is put into the instruction queue and scheduler, this occurs in the cycle after `ID`. It will remain in the instruction queue until it is chosen to execute. When it is chosen to execute the instruction is removed from the instruction queue and moved to the `RR` stage. An instruction is chosen if its source operands are available (or can be bypassed when needed) and if resources such as functional units (FP adder, etc) will be available.

In the `RR` stage physical register values are read from the physical register file using `rsVal` and `rtVal`.

Spring 2003 and earlier: The `Q` stage can refer to two things: being placed in the instruction queue or being removed from it. It is used both ways in the solutions before Fall 2003.

An instruction is placed in the instruction queue in the cycle after ID (unless there is a stall, which is very rare in the homework and exam problems).

An instruction is removed from the queue when it is ready to execute (its operands will be available in the next cycle). When it is removed from the queue it reads physical register values (`rsVal`, `rtVal`) from the physical register file.

Applies to all solutions: After `Q` (before Fall 2003) or `RR` (Fall 2003 and later) the instruction moves to a *functional unit*. The following functional units are common in the homework and exams: `EX`, integer and logic operations; `B`, branch resolution; `L1 L2` or `EA ME` load/store unit (discussed further below), `A1 A2...` floating-point add, `M1 M2...` floating-point multiply.

1.2.4 Stages: `EX`, `A1`, `M1`, etc. Arithmetic and Logical Functional Units

These work like their statically scheduled counterparts. By default a complete set of bypass connections is assumed. After exiting a functional unit the instructions go to writeback, `WB`.

1.2.5 Stage: `B`, Branch Resolution Functional Unit

This is used to determine if a branch is taken or not (the *outcome*). See `WB` for more on recovery.

1.2.6 Stages: `L1`, `L2`, `EA`, `ME` Load/Store Unit

Stage names in parenthesis were used before Fall 2003. In the `EA` (`L1`) step the effective address of the load or store is computed. For loads `ME` (`L2`) indicates that the load/store queue and, if necessary, the cache is being checked for the data. If the data is found the next segment is `WB`, otherwise no segment is shown until the data arrives at which time `ME` (`L2`) is shown again. Stores first write their data into the load/store queue in the `ME` (`L2`) stage, after which loads can read it. Data is written to the cache (or memory) when, and if, the store commits.

In the 2-way superscalar example below the `lw` hits the cache and `lh` misses the cache.

```
lw $t1, 0($t2)     IF ID Q  RR EA ME WB     # Cache hit
add $t3, $t1, $t4 IF ID Q        RR EX WB

lh $t1, 0($t2)        IF ID Q  RR EA ME           ME WB     # Cache miss.
add $t3, $t1, $t4     IF ID Q                     RR EX WB
```

1.2.7 Stage: WB, Writeback; WC, Writeback and Commit

The `WB` stage occurs after the last functional unit stage (`EX`, `A4`, `M6`, etc). The default assumption for all problems is that any number of write-backs can be performed per cycle. This assumption is unrealistic but it makes solving problems far less tedious.

The following occurs during writeback: The result is written to the physical register file. The status of the instruction (complete, and whether an exception occurred) is written to the reorder buffer.

If the instruction is a branch and if the ID map has been backed up, the outcome of the branch (taken, not taken) is compared to the predicted outcome. If they differ recovery starts. Recovery consists of copying the commit map to the ID map and restoring the free list. If the ID map has not been backed up recovery will wait until the branch commits.

If the processor can do commit and writeback in the same cycle, this is shown with a `WC`. This capability is assumed before Fall 2003, primarily to prevent the pipeline execution diagrams from getting too wide.

1.2.8 Stage: C, Commit

Commit occurs strictly in program order. By default the maximum number of instructions that can commit per cycle is equal to the decode width.

During commit the commit register map is updated and the incumbent is put back on the free list. Remember, the incumbent is **not** the physical register assigned to the committing instruction, it is the physical register assigned to the last instruction that wrote the same architected register. (See the example for the ID stage.)

1.2.9 Reorder Buffer, a.k.a. Active List

A queue holding *in-flight* instructions. Instructions always enter the ROB in `ID` and normally leave in `C` or `WC`. They can also be *flushed*, that is, *squashed en-masse*, as part of a recovery due to an exception or misprediction, or some other event.

An instruction updates its ROB entry during `WB`, writing a `1` into the complete field (`C`) and the exception code in the `X` field (a zero might indicate no exception).

If over a long enough period instructions are fetched faster than they are being committed the ROB will fill up, stalling `IF` and `ID`. This can affect the CPI in some problems, such as Spring 2003 Homework 6, Problem 1. On real systems the ROB is mostly full due to cache misses. (A load that misses the cache waits a long time and so it might reach the head of the ROB before getting its data. If so, commit stops and until the data arrives the ROB fills.)

1.2.10 Instruction Queue/Scheduler

A list of instructions that have not yet executed. (It's called a queue but it's not first-in/first-out.) The *scheduler* monitors which functional units are busy and which registers are ready. From this information it chooses instructions to start, they will enter the `RR` stage. (Before Fall 2003 `RR` shown as a `Q`.)

Real systems may use several schedulers and queues, each can send instructions to a subset of functional units. For example, there may be an integer instruction queue and a floating-point instruction queue.

1.2.11 ID Map

A table giving the latest physical register assigned to each architected register. Written in ID; the entry number that is written is the architected destination register (say, `t2`), that entry is written with the new physical register number, (say 92). Since it's written in the ID stage it reflects the part of the program that has passed through ID.

1.2.12 Commit Map

A table giving the latest physical register assigned to each architected register. Written in `C`; just like the ID Map, the entry number that is written is based on the architected destination register, that entry is written with the new physical register number. Since it's written in the commit stage it reflects the part of the program that has passed through commit.

1.2.13 Physical Register File

A table giving register values. It is written in the `WB` stage using, of course, physical register numbers.

1.2.14 Free List

A list of unused physical registers. An instruction removes a physical register from the list when it passes through ID. That physical register will hold the result of the instruction. If an instruction were proud of its creation (the result) then it would hate to see it tossed into the recycle bin (free list). It is very fortunate that instructions do not throw their creations into the free list, but instead *throw out their predecessor's creation*. (That is, they put the incumbent in the free list.)

1.3 Problems, By Type

Problem types are described below in roughly order of increasing difficulty.

1.3.1 Pipeline Execution Diagram

Show a pipeline execution diagram for the code fragment below running on the following system ... Solving this type of problem is a matter of knowing the steps for instruction execution.

Solution Tips: Be sure to check for dependencies. Remember that instructions execute as soon as their source operands are ready (unless there is not a free functional unit, which is rare for this type of problem). If there is a branch misprediction, determine whether the ID Map is backed up; if it is, recovery starts in WB, otherwise it starts during commit.

See the following final exam problems: Fall 2004 problem 3(c), Spring 2000 problem 3, Spring 2001 problem 2, 1998 problem 2, 1997 problem 2.

1.3.2 Complete ID Map and Other Tables

A program executes on a dynamically scheduled system using Method 3 as shown in the pipeline execution diagram below. Using the tables provided, show the changes to the ID Map, Commit Map, and Physical Register File. Also show when each instruction commits.

Most of these problems are easy to solve, or at least partially solve, using a blind (to what is going on) mechanical procedure. See the following final exam problems: Spring 2005 problem 2 (easy), Fall 2004 problem 1, Spring 2002 problem 1, Fall 2001 problem 1, Spring 2000 problem 2. In the Spring 2003 final exam Problem 1 the tables are partially filled and the question asks that they be completed and a program written consistent with their values.

1.3.3 Load/Store Unit

See Fall 2001 problem 4a.

1.3.4 Non-Standard Systems

Many of the problems above could be solved by memorizing rote procedures. To test true understanding some problems ask about an unusual (not covered in class) dynamically scheduled system. There are two examples of this sort of problem (so far). In the first (Fall 2003 problem 2) the hardware is defective; tables must be filled in showing the incorrect execution. In the second, Spring 2002 problem 2 (the later parts) asks about the execution of predicated instructions on a dynamically scheduled system.