Lecture 5 notes

Recall that what increases CPI above ideal is hazards. We have outline three categories, structural hazards, data hazards (caused because of true or artificially trigerred data dependencies) and control hazards (caused because of branches).

Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls

–Ideal pipeline CPI: measure of the maximum performance attainable by the implementation

–Structural hazards: HW cannot support this combination of instructions

–Data hazards: Instruction depends on result of prior instruction still in the pipeline

–Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

Theory of data dependencies

We have seen several ways to reduce stall cycles due to hazards. Some of these ways can be implemented in software, others in hardware. One key technique is both cases is the reordering of the execution of instructions, so that stall cycles are replaced with useful instruction execution cycles. Any technique which reorders instructions needs to make sure that reordering instructions does not violate the original instruction execution order specified by the program. In other words, we do not want instruction reordering to violate any dependencies which are imposed in the original program execution order.

There are 2 approaches to exploiting ILP, one relying on hardware to discover ILP dynamically and a second using software (more specifically the compiler) to perform program transformations, including instruction reordering, which can improve ILP.

Loop-level parallelization is a technique which verifies if the iterations of a loop are independent. More specifically, we need to check for any pair of instructions in the program order If 2 instructions are:

–parallel, i.e. they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls (assuming no structural hazards)

–dependent, they are not parallel and must be executed in order, although they may often be partially overlapped

$Instr_J$ is data dependent (aka true dependence) on $Instr_I$:

1.$Instr_J$ tries to read operand before $Instr_I$ writes it

2.or $Instr_J$ is data dependent on $Instr_K$ which is dependent on $Instr_I$

I: add r1,r2,r3

J: sub r4,r1,r3

If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped. Data dependence in instruction sequence, means data dependence in source code, means that the effect of original data dependence must be preserved. If data dependence is caused by a hazard in pipeline, it is called a Read After Write (RAW) hazard.

The HW/SW must preserve program order: we need to execute instructions in the order in which they would execute, if executed sequentially as determined by original source program. Dependences are a property of programs. The presence of dependence indicates potential for a hazard, but whether there is an actual hazard and the number of stall cycles due to the hazard is a property of the pipeline.

Data dependencies are important because:

1) they indicate the possibility of a hazard

2) they determine order in which results must be calculated

3) they set an upper bound on how much parallelism can possibly be exploited

The goal of the HW/SW is to exploit parallelism by preserving program order only where it affects the outcome of the program.

A name dependence is a dependence where 2 instructions use the same register or memory location, called a name, but there is no actual flow of data between the instructions associated with that name; there are 2 versions of name dependence:

If $Instr_J$ writes operand *before* $Instr_I$ reads it, example:
I: sub r4,r1,r3

J: add r1,r2,r3

K: mul r6,r1,r7
This dependence is called an "anti-dependence" by compiler writers. Note that this dependence is not "caused" by the program, but by potential reordering of the instructions during their executions. In the original program, the two instructions just reuse the register "r1". The situation could be easily resolved, if the second instructions could just use another unused register, e.g. register "r8". This is called register renaming, i.e. J "renames" "r1"to "r8". If an anti-dependence, it is caused by a hazard in the pipeline, called a Write After Read (WAR) hazard

If $Instr_J$ writes operand *before* $Instr_I$ writes it, we have an "output dependence" (in compiler terminology).Example

I: sub r1,r4,r3

J: add r1,r2,r3

K: mul r6,r1,r7

Output dependence also result because instructions reuse the name of register ("r1" in the example). If in this example we assume that the intention of the programmer is not to store r4+r2 in r1, then the add instruction could use a different target register, by renaming "r1" to, say "r8". If the output dependence is caused a hazard in the pipeline, called a Write After Write (WAW) hazard. In principle, instructions involved in a name dependence can execute simultaneously if name used in instructions is changed so instructions do not conflict. This can be done either by the hardware, or by the compiler.

We move on to describe control dependence: An insturction in a program may be control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program correctness. Here is an example:

```
if p1 {
        S1;
};
if p2 {
        S2;
}
```

In this example, S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1. In practice, we often allow hardware to violate control dependences and execute useless instructions, provided that we can later somehow undo the error, by preventing the wrong instructions from modifying architectural state.

Exceptions, in addition to data dependences, impose constraints in program execution order. The execution of instructions in the processor should not change how exceptions are raised in the program and should not raise new exceptions. Example:

```
        DADDU   R2,R3,R4
        BEQZ    R2,L1
        LW      R1,0(R2)
L1:
```

Assume in this examples that there is no branch delay slot. The LW may raise an exception if it accesses invalid memory. If we move the load before the branch to eliminate a potential stall between the DADDU and the BEQZ, we may throw an exception at a point where we are not supposed to throw an exception.

Loop unrolling

If the iterations of a loop are independent, we can exploit instruction-level parallelism by unrolling the loop. The key idea is to transform the loop, so that each iteration through the transformed loop executes several iterations of the original loop. We discuss a concrete example of unrolling and the required transformations of the code in class.

To apply unrolling, we need to understand how each instruction depends on another and how the instructions can be changed or reordered given the dependences. The process is as follows:

1.Determine if loop unrolling could be useful by finding that loop iterations were independent

2.Use different registers for each "iteration" in the body of the unrolled loop to avoid unnecessary constraints forced by using same registers for different computations

3.Eliminate the extra test and branch instructions and adjust the loop termination and iteration code. In particular, if we unroll a loop k times (i.e. put k iterations of the loop in the body), we need to execute the unrolled loop floor(n/k) times and pre-execute n mod k iterations, so that the loop is complete.

4.Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent A tricky aspect of unrolling is that the transformation requires analyzing memory accesses to find that they do not refer to the same address

5. Schedule the code, preserving any dependences needed to yield the same result as the original code

Unrolling, like most optimizations, does not come for free and has limitations. Unrolling reduces branch overhead, but the relative improvement it achieves diminishes as we unroll more. Further, unrolling creates register pressure, therefore the unrolling factor may be capped at a very low value by the limit of available registers.

Branch prediction is a technique used to reduce the impact of branches on the pipeline. Unrolling serves the same purpose specifically for loops. Branch prediction is a more general technique.

Branch prediction works because underlying programs have regularities. Regularities, repeatability most importantly, are to be found often in both the cde paths executed by a program and the data accessed by a program. Think loops in

scientific codes as an example. We have discussed some schemes for static branch prediction (e.g. predict always taken, or always not-taken) in class. Contrary to static branch prediction, dynamic branch prediction uses information about the dynamic behavior of branches, that is, the outcomes (Taken/NotTaken) of any given branch, as the program executes, to make predictions. Conventional wisdom is that dynamic branch prediction is better than static branch prediction.

The performance of a branch predictor is a function of the accuracy and the cost of misprediction. The performance is also affected if the introduction of the branch predictor affects the clock cycle.

A very simple predictor can be implemented with a Branch History Table (BHT). We are using some LSBs of the PC to index a table, with one entry for each group of branches that have the same LSBs in their PC. The entry says if the branch was taken or not taken last time. The problem with this scheme is that for a very simple loop, it can cause up to two mispredictions (one at the exit of the last iteration, and one in the first iteration, if the loop is traversed more than once in the code).

A solution hear is to track more history per branch and use hysteresis before changing the prediction from T to NT and vice versa. A history of the last two outcomes of the branch, and changing the prediction after two same outcomes (T-T, or NT-NT), solves the aforementioned problem with the loop. We implement these schemes using saturating counters.

There are two reasons of misprediction in the BHT. One is that the history that we are using may just not predict the right outcome of the branch (wrong guess). The other is that since multiple branches with the same LSBs in the PC can map to the same entry in the BHT, the BHT may give us a prediction for the wrong branch (not the one we are executing, but some other with the same LSBs).

Can we improve branch prediction?

Yes, if we exploit the idea of **correlation.** Instead of recording the history of each branch in isolation, we record the history of the *m* most recently executed branch instructions (can be the same or different branch instructions). We use the pattern of the recent branches to index the BHT. Each entry in the BHT uses a number of bits (say n) and the BHT uses an automaton to derive the prediction. This general scheme is called a (m,n) predictor (record last m branches, select between 2^m history tables, each with *n*-bit counters, use the n-bit counter to derive the prediction). We have explained (0,1) and (0,2) predictors. We show examples of (2,2) predictors in class. Correlation tends to improve significantly the performance of branch predictors.

Tournament predictors, use saturating counters to decide what *predictor* to use for prediction. Notice the difference with correlation predictors: In correlation predictors, the history of the branches serves as the index to the BHT. In tournament predictors, the history of the branch (and more specifically the correctness of predictions) selects a predictor. A common example, is the use of two

predictors, one with global history and one with local history. We can use a 2-bit saturating counter to switch between predictors upon two misprediction.

Figuring out the branch target: The branch involves both the decision and the calculation of the target address if the branch is taken. Branch target calculations needs resource in the pipeline and at least one cycle. It makes no sense to do predictions, unless we can also have the target of the branch during the IF stage, otherwise, we will stall for one cycle anyway. Solution: We use a Branch Target Buffer (BTB), which keeps the target addresses of branches already seen in the code. If a branch executes and its PC address is stored in the BTB, and if the predictor says the branch is taken, we use the BTB to retrieve directly the target of the branch.

Unfortunately, the BTB is a finite structure, and there may be conflicts between branches mapping to the same entries in the BTB.

Dynamic Scheduling

Dynamic instruction scheduling is a technique used by the hardware to rearrange the instruction execution to reduce stalls while maintaining data flow and exception behavior. Dynamic instruction scheduling is a powerful technique, because it handles cases when dependences are not known statically (e.g. at compile time).

Dynamic scheduling allows the processor to tolerate unpredictable delays such as cache misses, by executing other code while waiting for the miss to resolve. Further, it allows code compiled for one pipeline to run efficiently on a different pipeline. The technique also relieves the compiler from the burden of static dependence analysis.

Hardware speculation, a technique which attempts to predict future hardware events and improve the degree of instruction-level parallelism, builds on dynamic scheduling.

The key idea of dynamic scheduling is to allow instructions following a stall to proceed, assuming no dependencies are violated. For example:

```
DIVD   F0,F2,F4
ADDD  F10,F0,F8
SUBD   F12,F8,F14
```

Dynamic scheduling enables out-of-order execution and allows out-of-order completion of instructions (e.g., SUBD). In a dynamically scheduled pipeline, all instructions still pass through the issue stage in order (in-order issue). Beyond the issue stage, instructions can be reordered. Dynamic instruction scheduling may introduce WAR and WAW hazards (name dependencies), because of instruction reordering. This complicates hazard resolution in the processor. Further, instruction reordering may violate correct execution with respect to exceptions.

The first step to dynamic instruction scheduling is to identify as soon as we issue the instruction whether there is a structural hazard or not. We check whether the instruction has a free execution unit (e.g. adder, multiplier, etc.) to accommodate it.

The second step to dynamic instruction scheduling is to check whether the instruction has any pending unresolved data dependencies. We check if the operands of the instruction are "ready" (presumably produced earlier by functional units of the processor) or "not ready". In the latter case, the instruction needs to stall and wait for its operands.

We study Tomasulo's algorithm in class. The basic elements of the algorithm are:

- Functional units, called reservation stations, which are producing results for registers and memory. These results are also used to resolve pending data dependencies. Loads and stores have their own functional units.
- Registers used by instructions, which are replaced by "pointers" to reservation stations during instruction execution. The algorithm renames the registers used by the instruction to the execution units that will produce the input operands and store the result of the instruction.
- Results are forwarded between instructions not through registers, but through the reservation stations using a bus (called a "common data bus")
- Integer instructions can execute past branches, effectively implementing a "predict taken" scheme, which enables overlapping of iterations in loop.

Tomasulo's algorithm achieves performance improvement thanks to instruction overlap and out-of-order (OOO)execution, which reduces stalls. The key technique for OOO execution is register renaming. Instructions can issue past pending control flow operations and proceed up to the point where the pending dependence needs to be resolved. The algorithm distributes the hazard detection and resolution logic between functional units. The common data bus helps, in the sense that it can be used to trigger multiple instructions waiting for results simultaneously. The algorithm effectively eliminates stalls for WAW and WAR hazards. Unfortunately, there are also downsides: The algorithm has increased hardware complexity and needs many associative searches in tables to detect hazards between FUs. Furthermore, the CDB is a bottleneck. Lastly, the algorithm as is may introduce inopportune exceptions.