COMP4690
Computer Systems and Architecture

Dr. Rasit Eskicioglu
E2-574
rasit@cs.umanitoba.ca
Outline

• Introduction and Review
  – Design issues, technologies, tools

• Advanced ILP
  – Out of order execution, branch prediction, register renaming, value prediction

• Introduction to Parallel Architectures
  – Motivation, challenges, strategies

• Multiprocessor Architectures
  – Cache coherence, UMA, NUMA, COMA
Outline (cont’d)

• Clusters and their Interconnections
  – Programming support, interconnect fabrics
• High Performance Memory Systems
  – Bandwidth and latency, interleaving, PIM
• High Performance I/O and Storage
  – Parallel I/O, scalability
• Evolving Architectures
  – Multithreaded, cellular and custom architectures
ADVANCED ILP
Trends in ILP

• Deeper Pipelining
  – Compare the “textbook” 5 stage pipeline with the Pentium 4’s 20 stage pipeline
  – Deeper pipelines offer increased potential speedup but:
    • what if we incur hazards or load up the wrong instruction sequence?
      – very high overhead for restart
    • how do we keep the pipeline fed?
      – higher demand on memory system
Trends in ILP (cont’d)

• Superscalar execution
  – Lots of chip space so we can provide multiple functional units (FUs) but:
    • how do we find instructions to keep the functional units busy?
      – code characteristics (frequent branches)
      – more complex control (to manage FUs)
    • how do we feed the functional units?
      – higher demand on memory system, again
Alternatives to ILP

• We can also use available chip space in other ways to provide improved performance
  – Simultaneous Multi-Threaded (SMT)
  – Multi-core implementation

• Often we combine ILP with these as well
  – Just as we combine pipelining and superscalar
    • e.g., a multi-core processor where each core is pipelined superscalar, possibly supporting SMT
Deeper Pipelines

• Increasingly, the pipelines used in modern processors are becoming deeper
  – Known as “superpipelining”
• An obvious question to ask is “where do the new stages come from”?
• This must come from subdividing existing stages
  – Looking at execution time, there are two obvious candidate stages for subdivision:
    • EXecute and MEMory access
Deeper Pipelines (cont’d)

• The sort of pipelining we have looked at so far is sometimes referred to as “I-Unit” pipelining
  – Since we are pipelining instruction evaluation
• Another type of pipelining is also possible which is known as “D-Unit” pipelining
  – D-Unit pipelining pipelines the data operations
    • that is, the processing performed by the functional units
  – For complex, long running operations (e.g., floating point) D-unit pipelining can be effective
• Consider an example:
  – Floating point addition is composed of a sequence of operations that may be easily used as the individual stages of a FPADD pipeline
  – An FPADD might be performed in four steps:
    • exponent comparison
    • mantissa adjustment
    • mantissa addition
    • result normalization
Deeper Pipelines (cont’d)

- In exponent comparison, the difference (‘d’) between the two exponents is computed
- Then, in mantissa adjustment, the decimal point of the smaller number is right shifted ‘d’ places so the mantissa values may be added directly
- The two mantissas (adjusted) are then added
- Finally, if necessary, the result is normalized so there is only one digit to the left of the decimal point and the exponent is adjusted accordingly
Deeper Pipelines (cont’d)

- Example: Add
  - **Compare** exponents
    - $4 > 2$
  - **Alignment**
    - $2.900 \times 10^4$
    - $+ 0.099 \times 10^4$
  - **Add** mantissas
    - $2.999 \times 10^4$
  - **Normalize**
    - already normalized
Deeper Pipelines (cont’d)

• There is a potential for up to four times performance increase using pipelined FPADD
• Unfortunately, there is a critical difference between I-Unit pipelining and D-Unit pipelining
  – In I-Unit pipelining, there is a near constant supply of instructions to decode, fetch, etc.
    • subject to changes in control flow
  – With D-Unit pipelining, it is unlikely that there will be a constant supply of FPADD operations (or FPMUL or ...)
Deeper Pipelines (cont’d)

• This means that the full benefit of D-unit pipelining may be harder to achieve
• The effectiveness of D-Unit pipelining is heavily dependent on the instruction stream being executed
  – Many programs do a large number of floating point operations (e.g., on vectors and arrays), but they seldom code them in a sequence
    • we use loops instead
Deeper Pipelines (cont’d)

• Since operations on vectors are implemented using loops, the repeated operations occur in loop bodies
  – With intervening instructions to disrupt a pipeline
• One approach to providing the needed stream of identical operations is to use “loop unrolling” or “loop unravelling”
  – This can be done manually or using a parallelizing compiler
Deeper Pipelines (cont’d)

- We can unravel the loop as follows:
  
  \[
  
  etc.

- This removes the intervening loop control instructions thereby enabling the effective use of D-Unit pipelining
Deeper Pipelines (cont’d)

- To prevent code bloat (and associated potential problems with cache efficiency), unravelling can be done to a limited extent for large loops:

```plaintext
FOR I:=1 TO 250 BY 4 DO
  BEGIN
    A[I+0]:=B[I+0]+C[I+0];
    A[I+1]:=B[I+1]+C[I+1];
    A[I+2]:=B[I+2]+C[I+2];
    A[I+3]:=B[I+3]+C[I+3];
  END;
```
Deeper Pipelines (cont’d)

• Loop unravelling works well for loops with static (i.e., compile-time determinable loop bounds) but what about a loop like the following?

```plaintext
READ (N);
FOR I:= 1 TO N DO
  BEGIN
    A[I] := B[I]+C[I];
    END;
```

How many times do we unravel the loop?
Deeper Pipelines (cont’d)

• This is just one of a number of possible problems that may limit unravelling
• This one is easy to fix:
  – Pick a value ‘k’ which is, say, a small multiple of the D-unit pipeline size
  – Unravel in chunks of size ‘k’
  – Manage the unravelling with new loop code as follows:
Deeper Pipelines (cont’d)

READ(N);
FOR I:=1 TO (N DIV K)*K BY K DO BEGIN
    A[I+0]:=B[I+0]+C[I+0];
    A[I+1]:=B[I+1]+C[I+1];
    A[I+2]:=B[I+2]+C[I+2];
END;
FOR I:=(N DIV K)*K+1 TO N DO BEGIN
    A[I]:=B[I]+C[I];
END;

K=3 unravelled iterations

The leftovers
Deeper Pipelines (cont’d)

• In some cases, special instructions are added to a machines ISA to support operations on vectors
  – This is because such operations are very common in graphics, signal processing, etc.
  • consider: MMX instructions on Intel machines or ...
• Such instructions are great for D-unit pipes
  – E.g., VFPADD A, B, C, 10000
Deeper Pipelines (cont’d)

• Another pipeline stage that is a good target for subdivision is memory access
  – This is really cache access

• How can we pipeline cache access?
  – Perform directory search and cache RAM access in separate stages
  – Also make sure we have separate I and D caches so that we can do instruction fetch and memory load/store concurrently and pipelined
Superscalar Execution

• Just having multiple functional units does not enable the execution of more than one instruction at a time (a.k.a. *superscalar* execution)
• First, there must be multiple ready to execute instructions available
• This implies the need to have pre-fetched and decoded multiple instructions
  – Multiple, parallel pipelines
Superscalar Execution (cont’d)

- In addition, data dependences between instructions must be considered
  - Independent instructions can be done in parallel

- To ensure dependences are respected, hardware must be provided to track the use of operands and results in the concurrent instructions
  - Two common mechanisms for this are “Scoreboarding” and “Tomasulo’s Algorithm”
Superscalar Execution (cont’d)

- Abstractly, what must be done is to maintain the Read and Write sets of each instruction.
- The read/write set information is used to determine when instructions may execute concurrently and when instruction execution must be delayed.
  - E.g., stalls introduced to preclude reading stale data.
- To improve performance, these techniques are often combined with *forwarding*. 
Superscalar Execution (cont’d)

• Forwarding connects the result of one functional unit directly to the input of another
  – Avoiding the delay of a store followed by a load

• A chief drawback of the use of multiple functional units is the high cost of replicating the hardware to implement multiple D-units
  – Compared to pipelining, for instance

• A lower cost alternative is to use what are known as virtual functional units
Superscalar Execution (cont’d)

• Rather then replicating the actual functional unit hardware, FIFO buffers are placed on the inputs of each functional unit

• Consider the following:
Superscalar Execution (cont’d)

• Each buffer is addressable as if it were connected to a separate adder (or ...)
  – Forwarding may also be done directly to these buffers from the output of a functional unit
• Using virtual functional units does not increase the speed of execution of the instructions queued in the buffers (there’s only 1 adder)
Superscalar Execution (cont’d)

- What it does is allow us to pre-schedule instructions to get them “out of the way” so we may schedule subsequent independent instructions:

  - ADD \( R_1, R_2 \)
  - ADD \( R_3, R_4 \)
  - ADD \( R_5, R_6 \)
  - MUL \( R_7, R_8 \) ← can now be scheduled earlier
Superscalar Execution (cont’d)

• When we schedule multiple instructions in parallel on different functional units we are introducing *out-of-order execution*

• This is theoretically simple as long as dependences are respected but there are some subtleties to consider
  – E.g., what about interrupts?

• Let’s look at how to handle dependences
Superscalar Execution (cont’d)

• Scoreboarding, like Tomasulo’s algorithm, enables out of order execution thereby permitting better usage of functional units
• We will consider a register-only architecture to simplify the presentation
  – Modern architectures often include a memory unit as one of the available functional units to support concurrent loading/storing of operands/results together with other operations
Superscalar Execution (cont’d)

• During instruction decode, the scoreboard is updated to record information concerning data dependences between the instruction being decoded and those already executing.
• It can then choose when to schedule certain instructions based on that dependency information.
• Information on unit availability is also tracked.
Superscalar Execution (cont’d)

- Three tables are maintained (logically):
  - Instruction status table
  - Functional Unit status table
  - Register Result status table

- Consider the following example (from P.247 of Hennessy&Patterson, “Computer Architecture A Quantitative Approach, 2nd edition”)
Superscalar Execution (cont’d)

• This example assumes two multipliers, one adder, one divider, and an integer unit for memory references, branches, and integer ops.

Instruction Status Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>ReadOps</th>
<th>Exec’d</th>
<th>WriteResult</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F6,34(R2)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LD</td>
<td>F2,45(R3)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MULTD</td>
<td>F0,F2,F4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SUBD</td>
<td>F8,F6,F2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DIVD</td>
<td>F10,F0,F6</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ADDD</td>
<td>F6,F8,F2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Superscalar Execution (cont’d)

### Functional Unit status table

<table>
<thead>
<tr>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>( F_i )</th>
<th>( F_j )</th>
<th>( F_k )</th>
<th>( O_j )</th>
<th>( O_k )</th>
<th>( R_j )</th>
<th>( R_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Yes</td>
<td>Load</td>
<td>( F_2 )</td>
<td>( R_3 )</td>
<td>( F_4 )</td>
<td>Int</td>
<td>Int</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Mult1</td>
<td>Yes</td>
<td>Mult</td>
<td>( F_0 )</td>
<td>( F_2 )</td>
<td>( F_4 )</td>
<td>Int</td>
<td>Int</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Mult2</td>
<td>Yes</td>
<td>Sub</td>
<td>( F_8 )</td>
<td>( F_6 )</td>
<td>( F_6 )</td>
<td>Mult1</td>
<td>Int</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Add</td>
<td>Yes</td>
<td>Div</td>
<td>( F_{10} )</td>
<td>( F_6 )</td>
<td>( F_6 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divide</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( Q_j \) and \( Q_k \) are the operand source register specifications & \( R_j \) and \( R_k \) are the operand-ready flags.

### Register Result status table

<table>
<thead>
<tr>
<th>( F_0 )</th>
<th>( F_2 )</th>
<th>( F_4 )</th>
<th>( F_6 )</th>
<th>( F_8 )</th>
<th>( F_{10} )</th>
<th>( F_{12} )</th>
<th>etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mult1</td>
<td>Int</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Sub</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Div</td>
</tr>
</tbody>
</table>
Superscalar Execution (cont’d)

• Such dynamic instruction scheduling mechanisms also introduce the possibility of out of order instruction completion and this can create additional problems.
• For example, what if we re-order two divide operations and then get a divide by zero exception?
  – Debugging in such an environment can be painful if we don’t have hardware to undo instructions that shouldn’t have completed.
Superscalar Execution (cont’d)

- **Tomasulo’s algorithm** is a method for tracking data dependencies between concurrently executing instructions originally used in the IBM 360/91 (c. 1967)
- The 360/91 had multiple floating point units and Tomasulo’s algorithm was employed to permit concurrent execution between them
  - By ensuring data dependences were respected
Superscalar Execution (cont’d)

• The 360/91 used virtual D-units and a form of forwarding which are an integral part of Tomasulo’s algorithm
• Rather than centralizing responsibility for routing data values to where they are needed, Tomasulo’s algorithm has the destinations themselves monitor a “result bus” to collect the data they need
• A tagging scheme is used to enable this
Superscalar Execution (cont’d)

• Each instance of a data value is assigned a tag value which is unique throughout the corresponding instruction’s operation
• Operations are scheduled onto available units via a queue of “reservation stations” together with the tag information for their operands and result value
• When a new value is produced it is placed on a common result bus along with its tag
Superscalar Execution (cont’d)

• Any unit or reservation station waiting for a given value monitors the result bus and when it sees a matching tag takes the corresponding data value
• Operand values are thus collected by the “users” of the operands when they are produced by executing instructions
• When all necessary operands are collected a new operation is scheduled
Superscalar Execution (cont’d)

• A different approach to tracking the data relationships between dynamically scheduled instructions is the use of *scoreboarding* techniques
  – These were pioneered in the CDC-6600 at approximately the same time as Tomasulo’s algorithm was developed for the 360/91

• Scoreboarding is a centralized approach with the “scoreboard” recording all necessary resource usage information
Supporting ILP

• As ILP becomes more and more aggressive we need to apply several techniques (both H/W and S/W) for it to be effective
• We will look at several of these including:
  – Branch prediction
  – Trace scheduling
  – Speculative execution
  – Register renaming
  – Value prediction
  – Software pipelining
Supporting ILP (cont’d)

• The general theme behind all of these techniques is keeping the CPU hardware (pipelines, functional units, etc.) busy
  – E.g., branch prediction and trace scheduling try to avoid delays due to incorrectly fetched instruction stream(s), register renaming tries to avoid unnecessary data dependences, and software pipelining attempts to improve code characteristics
Branch Prediction

• We need to provide a constant stream of instructions to feed our CPU
• If we encounter a conditional branch in the instruction stream we must decide whether the branch will be taken or not
  – And fetch and decode instructions along the selected execution path
• If we guess wrong then we will have delays
  – E.g., control hazard → stall in a pipeline
Branch Prediction (cont’d)

• There are two approaches that can be taken in such situations:
  – Try to predict whether or not the branch will be taken and fetch, decode, etc. along that path
    • great if we are right but no help when we are wrong
  – Fetch and decode along both paths so we are ready to execute regardless of the outcome
    • costly in terms of hardware particularly for sequences containing many nearby conditional branches

• We will focus on branch prediction
Branch Prediction (cont’d)

• What we need to be able to do is to accurately predict the outcome of branches so we can pre-fetch along the correct path
• This can be done conservatively in software (i.e. static branch prediction) or in hardware (i.e. dynamic branch prediction)
• Static branch prediction is done by the compiler – Which builds and examines a control flow graph
Branch Prediction (cont’d)

• A natural approach to predicting branches in a compiler is to predict that back edges will be taken
  – This is logical since back edges almost always represent a branch to the top of a loop and loops normally execute their bodies (and hence the branch) more than once

• The question is how to communicate this information to the hardware
Branch Prediction (cont’d)

• This requires some sort of cooperation between the architecture and the compiler
  – E.g., a BranchPredict bit associated with each branch instruction which is set by the compiler and interpreted by the hardware during instruction decode (ID)

• Question: “If we need hardware cooperation anyway, why bother with static prediction?”
Branch Prediction (cont’d)

• The general answer is that we don’t bother with static prediction
  – Even though we can extend static branch prediction to use profile history information

• In most cases, dynamic branch prediction does as well or better than static branch prediction
  – The rest of our discussion will therefore consider dynamic branch prediction
Branch Prediction (cont’d)

• One of the simplest approaches to dynamic branch prediction is the use of a branch history table (BHT)
  – A.k.a a branch prediction buffer
• Such a table is indexed by some function of the address of a branch instruction (commonly the low order bits) and returns a Taken/Not-Taken (T/NT) result
  – I.e., a single bit is stored to indicate whether or not the branch is predicted as taken
The bit returned is used to determine the direction of pre-fetching done

If, after execution, the bit is *incorrect* then it is inverted in the table

– Thus, mis-predictions are intended to be self-correcting

Initially branches are marked as taken

– Most branches in code are taken, so ...

A fundamental limitation of this technique is the indexing function
Branch Prediction (cont’d)

• An ideal branch predictor would have an entry in the table for every possible branch
  – I.e., would be indexed by the full address

• This is clearly impractical as only a small table is possible

• Hence, a subset or mapping of the branch’s address must be used
  – For a table of \( N \) entries, we will consider using the low order \( \log_2 N \) bits of the address
Branch Prediction (cont’d)

• The effect of using a subset of the bits is that “collisions” may occur
  – Two branches may occur at addresses which map to the same entry in the branch history table
• This means that we may be using the recent behavior of some branch $B_x$ to predict the outcome of a different branch $B_y$
  – Probably not a good idea :-)
Branch Prediction (cont’d)

• Consider a 16 entry BHT operating on the following code segment within a loop

\[
\begin{align*}
&0100_{16} & \text{TST} & \text{R1} \\
&0120_{16} & \text{BEQ} & \text{L1:} \\
&\text{...} & & \\
&0120_{16} & \text{BGT} & \text{L44:}
\end{align*}
\]

– Both branches map to the same location so the behavior of the BEQ affects the prediction for the BGT

• this is because \(0100\) and \(0120\) are both congruent to \(0\) mod 16
Branch Prediction (cont’d)

- This simple technique is known as one-bit prediction and is not bad but also not as good as we would like
  - E.g., for an *inner* loop which is executed N times, it mis-predicts twice instead of only once as with static prediction
    - this is due to the toggling of the prediction bit after the mis-prediction at the end of the loop’s execution
    - this guarantees a mis-predict at the start of the next run through the loop
Branch Prediction (cont’d)

• To address this problem, more bits may be added to each entry in the history table
• This allows the scheme to toggle prediction after ‘n’ mis-predictions instead of one
  – In the preceding example this avoids the extra mis-prediction associated with nested loops
• Interestingly, it turns out that a 2-bit predictor is about as much as you need
  – Adding more bits doesn’t buy us much
Branch Prediction (cont’d)

• Such simple predictors do not, however, offer ideal branch prediction capabilities
  – Complex integer programs with large numbers of branches may still see upwards of 10% of all branches mis-predicted
    • since the cost of mis-prediction will be high, a 10% failure rate will be costly

• Some programs perform branches based solely on input data
  – We can’t hope to predict these
Branch Prediction (cont’d)

• What these simple predictors do miss that we can predict is *correlated branches*
  – That is, situations where the outcome of one branch effectively determines (with high probability) the outcome of another
  – These are surprisingly common in many programs

• To address this *two-level* branch prediction schemes were developed
Branch Prediction (cont’d)

• The idea is to use the outcome of previous branches to select between a number of possible branch predictors
  – E.g., if the last branch was taken, then use this prediction bit otherwise use a different prediction bit

• In general, this defines a family of possible two-level predictors known as \((m,n)\) predictors
Branch Prediction (cont’d)

• An \((m,n)\) predictor uses the behavior of the last ‘m’ branches to select one of \(2^m\) ‘n’ bit branch predictors
  – I.e., based on the T/NT pattern of the preceding branches we select a particular predictor for the current branch

• Two-level predictors significantly improve prediction accuracy at very low cost
  – Global history is just an ‘m’ bit shift register
Branch Prediction (cont’d)

- The shift register contents (i.e., global branch history information) is combined with the low order bits of the branch address to index into a table of branch prediction bits
  - This is most easily viewed as a 2D table although this, of course, is not the implementation structure used
Branch Prediction (cont’d)

– Consider:

3-bit branch address

2-bit global branch history
Branch Prediction (cont’d)

• An alternative approach modifies the basic two-level scheme to produce what are referred to as 
  *elastic history buffers*

• This scheme recognizes that different branches may have different degrees of correlation
  – Some branches need to look at two previous branches, others need three, etc.
Branch Prediction (cont’d)

• This simple extension yields, on average, 25% improvement in branch prediction accuracy over the fixed correlation scheme (just presented)
• The idea is to characterize the required degree of correlation using a profiling run of the program, provide this information to the compiler, and have the compiler encode the desired degree in the branch opcode
Branch Prediction (cont’d)

- This technique requires space in the branch opcode field which may or may not be available.
- It is restricted to considering the last ‘k’ branches taken.
  - Where ‘k’ is the degree of correlation.
- It leaves some history buffer entries unused.
- It is, of course, also dependent on the accuracy of the profile run.
Branch Prediction (cont’d)

• Branch prediction is a very useful tool in enabling superscalar and heavily pipelined execution

• In certain cases, however, predicting a branch during instruction decode (ID), when we first know we are dealing with a branch instruction, is too late
  - Pipeline stalls or other delays may occur
Instead of just knowing that a branch will or will not be taken, we can do better if we know the destination address of the branch.

- Or perhaps even better if we have the instruction(s) at that address available.

This leads to a modification of the dynamic branch prediction schemes we have seen to consider “branch target buffers” instead of branch prediction buffers.
Branch Prediction (cont’d)

• A branch target buffer stores pairs
  \(<\text{BranchAddr}, \text{TargetAddr}>\)
  – The \text{BranchAddr} field is used as a key to look up corresponding \text{TargetAddr} values
  – When we encounter a branch instruction during decode, we lookup the associated target address
    • the address of the target instruction, if the branch is taken
  – This immediately gives us the “next PC” to fetch from
Branch Prediction (cont’d)

• In essence, the branch target buffer is really a branch target *cache*
• If we can quickly determine if the needed entry is in the cache, we win (assuming the branch is taken)
  – Probably a smallish “associative” cache
• A cache with exact match is necessary to prevent inappropriate control flow
Branch Prediction (cont’d)

• Note that we only need to store branches that are predicted taken in the branch target buffer, since a not taken branch simply falls through which is the normal next-PC

• Under ideal circumstances, we can achieve a branch penalty of zero using branch target buffers when we have a buffer “hit”

• When we “miss”, the penalty may be high
  – Extra overhead managing the buffer
    • as well as handling the mis-prediction
Branch Prediction (cont’d)

• Things are more difficult with multi-bit prediction
  – Do we store multiple target addresses?
    • expensive!
  – This is commonly solved by using both a prediction buffer and a target buffer

• The idea of a target buffer can be taken one step further as well
Branch Prediction (cont’d)

• If getting the target *address* faster is good, surely, getting the target *instruction* itself is even better
• Instead of storing the target address in the buffer, store the instruction at that address
• This can lead to zero cycle branches
  – During IF, we match address in target buffer and then forward the cached target instruction for ID
Value Prediction

• Predicting branches based on past behavior is a common approach but not the only one.
• If we can predict the *value* of variables upon which branch outcomes are based, then we can know, with certainty, whether or not a branch will be taken.
  – Value prediction is a general technique and can be used for more than branch prediction.
Value Prediction (cont’d)

• The basic idea is that, in addition to temporal and spatial locality, there is also value locality.
• In other words, a given location will tend to have the same (or a predictable) value during a given time period in the execution of a program.
  – E.g., a register might be assigned the values 1, 2, 3, 4, 5, ... so we predict it will get the value 6.
Value Prediction (cont’d)

- In practice, value prediction is used to predict the values of registers and this is useful in branch prediction
  - E.g., loops commonly count down to zero so if we can tell the register involved in the loop is going to zero we can avoid a mis-prediction
- We are most interested in knowing that registers will be updated in stride patterns
  - E.g., 1, 2, 3, 4, ... or 1, 3, 6, 9, ... or ...
Value Prediction (cont’d)

• To predict a basic stride variable, we might be able to use:

\[ v_n = v_{n-1} + (v_{n-1} - v_{n-2}) \]

• Value prediction can also be used for other purposes (though we won’t discuss them)
  – E.g., *Load* value prediction might allow us to bypass slow memory accesses
    • lower latency loads & decreased memory bus traffic
Value Prediction (cont’d)

• In general, value prediction may be done in software (statically/at compile time) or in hardware (dynamically/during execution)

• Ultimately, as with branch prediction, static prediction can exploit program knowledge...
  – E.g., loop upper bounds

• ... while dynamic prediction can exploit runtime knowledge
  – E.g., discover irregular value patterns
Speculative Execution

• Whenever we allow the hardware to execute instructions that we do not know for certain should be executed, that execution is said to have been *speculative*
  – E.g., whenever we predict a branch outcome and actually *execute* an instruction before we know whether the prediction was correct or not

• This allows us to avoid stalls in a pipeline, etc., but we must be able to undo the effects of speculatively executed instructions
Speculative Execution (cont’d)

• Speculative execution is useful when there is no other work to do
• If there is something non-speculative that can be done, it should always be done first
• Speculative work is done “optimistically” to avoid unused H/W (functional units, etc.)
• Further, it is only useful if the cost of undoing incorrect executions is less than the benefit offered by speculation
Trace Scheduling

• As we know, particularly in integer programs, there are many branches
• In such cases, branch prediction can be difficult and/or costly
• If we are able to support speculative execution, there is another alternative to branch prediction as we have seen it
• A compiler can “predict” how control will flow through a sequence of branches
Trace Scheduling (cont’d)

• Such a series of branches and associated instructions is known as a trace
• If we remove the branch related instructions from such a code sequence, we have a larger number consecutive instructions which can be more effectively scheduled
  – E.g., to keep a pipeline full or multiple functional units busy
Trace Scheduling (cont’d)

• Implemented in two stages:
  – *Trace Selection*
    • find a sequence of basic blocks (*trace*)
  – *Trace Compaction*
    • reorder code based on available resources
    • need “undo” code in case prediction is wrong

• Provides speedup only if the static prediction is good
  – Otherwise, undo code cancels benefit
More Support for ILP

• So far we’ve considered techniques focused on eliminating control hazards
  – I.e., disruptions to the flow of instructions into the processor for execution
• Structural hazards, of course, are addressed by adding more hardware
• A good question to ask is: “What can we do about data hazards?”
  – We need to think more about dependences
Dependences

- There are actually three types of dependences
  - So far we only considered one type
- We will define these dependences in terms of fine grained (instruction level) operations but they apply equally at larger scales
- The three types of dependences are:
  - True dependences,
  - Anti dependences, and
  - Output dependences
A true (flow) dependence occurs when one statement assigns a value to a variable and a subsequent statement uses it.

\[ S_1: a := b + c \]
\[ S_2: d := a/2 \]

The execution of \( S_1 \) must precede the execution of \( S_2 \).

We denote true dependence as:

\[ S_1 \triangleright S_2 \]
Dependences (cont’d)

• An *anti dependence* occurs when a variable is referenced in one statement and then is subsequently written by another statement.

  \[ S_1: \quad a := b + c \]
  \[ S_2: \quad b := d/45 \]

• The execution of \( S_1 \) must again precede the execution of \( S_2 \).

• We denote anti dependence as:

  \[ S_1 \rightarrow \partial \ S_2 \]
Dependences (cont’d)

- An **output dependence** occurs when a variable is written in one statement and then subsequently rewritten by another statement
  
  \[
  S_1: \quad a := b + c \\
  S_2: \quad a := d
  \]

- The execution of \( S_1 \) must again precede the execution of \( S_2 \)

- We denote output dependence as:
  
  \[ S_1 \preceq S_2 \]
• The first and most important thing to realize about dependencies is that only true dependencies represent actual data flow
• The others, anti and output dependencies, do not correspond to data flow
  – These dependencies are sometimes referred to as *name dependencies*
    • the dependence arises only due to re-use of the variable name (or address or register)
Register Renaming

• This observation leads to a fundamental compiler-based optimization technique known as register renaming.

• Consider the code resulting from a compiler “unrolling” or “unraveling” a loop:
  – Each iteration in the resulting loop body reuses the same registers.
  – This results in many name dependencies which may lead to a near-serial instruction schedule.
Register Renaming (cont’d)

- A compiler performing loop unrolling can also “rename” the registers (i.e., use different registers) to eliminate the name dependencies.
- This, in turn, allows more freedom in the H/W scheduling of the instructions within the new, unrolled loop body.
- Consider the following example (from Hennessy & Patterson, 2ed).
Register Renaming (cont’d)

• Unrolled code before register renaming:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>F0, 0(R1)</td>
</tr>
<tr>
<td>ADDD</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>SD</td>
<td>0(R1), F4</td>
</tr>
<tr>
<td>LD</td>
<td>F0, -8(R1)</td>
</tr>
<tr>
<td>ADDD</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>SD</td>
<td>-8(R1), F4</td>
</tr>
<tr>
<td>LD</td>
<td>F0, -16(R1)</td>
</tr>
<tr>
<td>ADDD</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>SD</td>
<td>-16(R1), F4</td>
</tr>
<tr>
<td>LD</td>
<td>F0, -24(R1)</td>
</tr>
<tr>
<td>ADDD</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>SD</td>
<td>-24(R1), F4</td>
</tr>
<tr>
<td>SUBI</td>
<td>R1, R1, #32</td>
</tr>
<tr>
<td>BNEZ</td>
<td>R1, Loop</td>
</tr>
</tbody>
</table>

What are the data dependencies?

original iteration
Register Renaming (cont’d)

• Code after register renaming:

```
Loop:
LD   F0,0(R1)
ADDD F4,F0,F2
SD   0(R1),F4
LD   F6,-8(R1)
ADDD F8,F6,F2
SD   -8(R1),F8
LD   F10,-16(R1)
ADDD F12,F10,F2
SD   -16(R1),F12
LD   F14,-24(R1)
ADDD F16,F14,F2
SD   -24(R1),F16
SUBI R1,R1,#32
BNEZ R1,Loop
```

What are the data Dependencies now?
Register Renaming (cont’d)

- Code after register renaming:

```
Loop:       LD   F0,0(R1)
            ADDD  F4,F0,F2
            SD    0(R1),F4
            LD    F6,-8(R1)
            ADDD  F8,F6,F2
            SD    -8(R1),F8
            LD    F10,-16(R1)
            ADDD  F12,F10,F2
            SD    -16(R1),F12
            LD    F14,-24(R1)
            ADDD  F16,F14,F2
            SD    -24(R1),F16
            SUBI  R1,R1,#32
            BNEZ  R1,Loop
```

Which instructions could be executed in parallel?
Register Renaming (cont’d)

• Code after register renaming:

Loop:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F0, 0(R1)</td>
</tr>
<tr>
<td>ADDD</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>SD</td>
<td>0(R1), F4</td>
</tr>
<tr>
<td>LD</td>
<td>F6, -8(R1)</td>
</tr>
<tr>
<td>ADDD</td>
<td>F8, F6, F2</td>
</tr>
<tr>
<td>SD</td>
<td>-8(R1), F8</td>
</tr>
<tr>
<td>LD</td>
<td>F10, -16(R1)</td>
</tr>
<tr>
<td>ADDD</td>
<td>F12, F10, F2</td>
</tr>
<tr>
<td>SD</td>
<td>-16(R1), F12</td>
</tr>
<tr>
<td>LD</td>
<td>F14, -24(R1)</td>
</tr>
<tr>
<td>ADDD</td>
<td>F16, F14, F2</td>
</tr>
<tr>
<td>SD</td>
<td>-24(R1), F16</td>
</tr>
<tr>
<td>SUBI</td>
<td>R1, R1, #32</td>
</tr>
<tr>
<td>BNEZ</td>
<td>R1, Loop</td>
</tr>
</tbody>
</table>

Why not let the compiler reorder the Instructions?
Software Pipelining

• An alternative to loop unrolling is a concept known as **software pipelining**
  – Sometimes called “symbolic loop unrolling”

• This technique also aims to increase the number of potentially concurrent instructions within a loop iteration
  – To improve performance via superscalar exec.

• The difference is the instructions selected for potentially concurrent execution
Software Pipelining (cont’d)

• The idea is to reorganize the loops so that each iteration in the modified loop consists of instructions from different iterations of the original loop
  – This is done without unrolling the loop so the result is not larger loop bodies but rather loop bodies that consist of more “compatible” instructions
    • those that may be more freely executed in parallel
Software Pipelining (cont’d)

- Software pipelining is best understood by considering an example and looking at the dependencies in an unrolled version of the loop to determine what should be in each new iteration
  - Hence, the term *symbolic* loop unrolling
- As with real pipelining, there will be some startup and shutdown overhead
Software Pipelining (cont’d)

• Consider the following loop (which is the same one we unrolled earlier):

  Loop:
  - LD F0,0(R1)
  - ADDD F4,F0,F2SD 0(R1),F4
  - SUBI R1 R1 #8
  - BNEZ R1,Loop

  Loop body
  Loop control

• Now, let’s unroll the loop so that we can figure out the inter-iteration dependencies (symbolically)
  – Three loop body statements so unroll 3 times
Software Pipelining (cont’d)

Iter i:
- LD F0,0(R1)
- ADDD F4,F0,F2
- SD 0(R1),F4

Iter i+1:
- LD F0,0(R1)
- ADDD F4,F0,F2
- SD 0(R1),F4

Iter i+2:
- LD F0,0(R1)
- ADDD F4,F0,F2
- SD 0(R1),F4

- We now take the selected instructions and construct the new (pipelined) loop body using them
Software Pipelining (cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F0,0(R1)</td>
</tr>
<tr>
<td>ADDD</td>
<td>F4,F0,F2</td>
</tr>
<tr>
<td>LD</td>
<td>F0,−8(R1)</td>
</tr>
<tr>
<td>SD</td>
<td>0(R1),F4</td>
</tr>
<tr>
<td>ADDD</td>
<td>F4,F0,F2</td>
</tr>
<tr>
<td>LD</td>
<td>F0,−16(R1)</td>
</tr>
<tr>
<td>SUBI</td>
<td>R1,R1,#8</td>
</tr>
<tr>
<td>BNEZ</td>
<td>R1,Loop</td>
</tr>
<tr>
<td>SD</td>
<td>0(R1),F4</td>
</tr>
<tr>
<td>ADD</td>
<td>F4,F0,F2</td>
</tr>
<tr>
<td>SD</td>
<td>−8(R1),F4</td>
</tr>
</tbody>
</table>

Loop:

- This code assumes anti-dependencies can be handled in hardware
  - E.g., read register in first half of cycle and write in second
Software Pipelining (cont’d)

- Software pipelining has one potentially significant advantage over loop unrolling.
- With loop unrolling, there is a large space overhead in terms of extra instructions:
  - This may have a detrimental effect on instruction cache performance.
  - Some code is also added for the startup and shutdown sequences in software pipelining but it will be much less for a heavily unrolled loop.
With all these compiler based techniques for enhancing ILP, one might wonder how far you could take the concept.

An interesting earlier concept that considered this idea was known as Very Long Instruction Word (VLIW) processing.

- Partially adopted in some real machines
  - e.g. Intel’s Itanium processor

**VLIW**

**EPIC:** Explicitly Parallel Instruction Processing
The idea behind VLIW was to make instruction scheduling a job for the compiler
  – Instead of adding complex hardware
    • e.g., scoreboard, Tomasulo’s algorithm, etc.

This allows the compiler to exploit its knowledge of the code structure to maximum benefit
  – Of course, sacrificing runtime knowledge

Saved chip space can also be repurposed
The name comes from the format of instructions on such a machine.

Consider:

Each long instruction word contains many “conventional” instructions.
Combining Support for ILP

- Loop unrolling, software pipelining and trace scheduling may be combined
- So may various forms of branch prediction, value prediction, speculative execution and trace scheduling
- This offers the potential for further performance improvements but if also makes for challenging design choices
  - And lots more of those lovely tradeoffs
Alternatives to ILP

• ILP improves the performance of the execution of a single program
  – By executing its individual instructions rapidly
• In most programming environments, there are actually several programs and/or threads available to execute at the same time
• We provide the illusion of a dedicated machine to each by quickly swapping between them (time sharing)
Alternatives to ILP (cont’d)

• With many processes/threads available to execute the question is: “Should we focus only on the performance of a single one?”
  – E.g., we could use the available silicon area to allow more than one thread to run concurrently instead of making one run faster
    • might be better/faster/easier/cheaper than context switching between them

• The answer really depends on our goals
Alternatives to ILP (cont’d)

• There are a number of alternatives to ILP that use the available chip real estate differently

• We will discuss two common ones:
  – SMT: Simultaneous MultiThreading, and
  – Multi-core processors

• Both reflect the need in “traditional” computing to have many threads run quickly
  – goal is to have responsive systems
Hardware Multithreading

- Hardware Multi Threading supports the execution of multiple threads concurrently
  - In essence, we make use of the available processor hardware (pipeline, functional units, etc.) to concurrently run instructions from more than one thread

- Depending on how hardware multithreading is done, this may offer several advantages
Hardware Multithreading (cont’d)

• To discuss hardware support for threads, we need to think about what a thread is.
• From the hardware perspective, a thread is an execution context:
  – PC, register values, etc.
    • i.e., the execution state
  – A thread also has an associated address space.
• The details of a thread affect how the HW manages threads:
  – Remember that threads used to be SW entities.
Hardware Multithreading (cont’d)

• For threads to be supported in hardware, they must be made known to the machine
  – More H/W and S/W interface stuff

• It then becomes possible for the processor to fetch instructions from more than one thread and have them available to schedule
  – This assumes, for the moment, that the threads come from the same address space (AS)
    • different ASs is harder to handle and may have detrimental performance side effects (TLB hit rate)
Hardware Multithreading (cont’d)

• To be able to effectively support multithreading, additional processor resources are clearly required

• Most notable among these are the need for multiple program counters and register sets, one per hardware thread
  – It may be possible to share registers (some form of hardware, thread-based renaming) but the total number of registers must increase
Hardware Multithreading (cont’d)

What resources are shared?

Looks like two processors!
Hardware Multithreading (cont’d)

• In a basic thread-aware processor, it is common to either:
  – Alternate between scheduling instructions from different threads (round robin fashion) ...
    • interleaving thread execution
  – ... Or switch to a different thread when the current one becomes blocked
    • e.g., due to a cache miss or ...
    • this provides some measure of latency tolerance
Hardware Multithreading (cont’d)

• In *Simultaneous MultiThreading* (SMT) (a.k.a. “hyperthreading” in Pentium) it is possible to instead **concurrently** schedule instructions from more than one thread onto the available hardware

• More complex but this results in much better utilization of processor hardware
  – E.g., if the FP multiplier is not used by the instructions from thread 1, we might find a use for it in the instructions from thread 2
### Hardware Multithreading (cont’d)

<table>
<thead>
<tr>
<th>SS</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th></th>
<th>X</th>
<th>X</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SMT</th>
<th>Y</th>
<th>X</th>
<th>V</th>
<th>X</th>
<th>Z</th>
<th>X</th>
<th>V</th>
<th>Z</th>
<th>Y</th>
<th>Z</th>
<th>X</th>
<th>X</th>
<th>V</th>
<th>Z</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>Z</td>
<td>Y</td>
<td>X</td>
<td>V</td>
<td>Y</td>
<td>X</td>
<td>V</td>
<td>Y</td>
<td>X</td>
<td>Z</td>
<td>V</td>
<td>V</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>Z</td>
<td>Y</td>
<td>V</td>
<td>X</td>
<td>V</td>
<td>X</td>
<td>Y</td>
<td>X</td>
<td>V</td>
<td>Y</td>
<td>Y</td>
<td>V</td>
<td>V</td>
<td>X</td>
</tr>
<tr>
<td>Y</td>
<td>X</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>X</td>
<td>V</td>
<td>Z</td>
<td>Z</td>
<td>V</td>
<td>V</td>
<td>X</td>
<td>Z</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- In practice, the degree of simultaneous multithreading is often limited
  - Typically to two threads with current technology

**Much better resource utilization!**
Multi-core

• Another alternative to aggressive ILP that is becoming very popular is the use of multi-core processors

• In a multi-core processor there are separate execution cores
  – I.e., the execution resources are not shared

• This means there are no hardware conflicts between threads but what about utilization?
Multi-core (cont’d)

• Resource utilization is less of a concern with multi-core chips
  – Two generic processes are running anyway

• Given a chip/die of the same size, if you want to have two cores, they must be simpler than a single aggressive ILP core
  – This means that there will be less resources to go to waste in each core anyway
Multi-core (cont’d)

- Consider the following block diagram of a dual-core Opteron processor
  - From an online LLNL tutorial

- Note the crossbar and “Hyper Transport” in support of shared memory access (more later in the course!)
Multi-core (cont’d)

- There are many possible variations to the basic multi-core architecture and its surrounding hardware
  - E.g., dual core Xeon has a shared L2 cache while the dual core Opteron has separate L2s

- Current technology in general purposes multi-core processors supports dual and quad core processors but this number per chip will grow
  - E.g., Intel has an 80 core processor in the lab
    - do you think these are aggressive cores?
    - what will happen to memory bandwidth with 80 cores?
Multi-core (cont’d)

• There are other types of multi-core processors
  – Sun’s Niagara family is an example of a more aggressive general purpose multi-core processor
    • Niagara 1: 8 simple cores running 4 threads each
    • Niagara 2: 8 cores running 8 threads each
  – IBM’s Cell processor includes different cores
    • 1 PowerPC and 8 simple Synergistic PUs
  – Multi-core designs have also been used (for some time) in special purpose machines
    • E.g., Network and Graphics Processors
Choosing Technologies

- The bottom line is that for a given fabrication technology, there is a limited number of transistors you can fit on a die.
- This means you must choose between multi-core, SMT and the various forms of ILP and their combinations to make a processor to meet your necessary target market.
- Further, costs dictate that there will be few target markets so special purpose processing will have to be done with available HW.