Code Scheduling

– Wilhelm/Maurer: Compiler Design, Chapter 12 –
Mooly Sagiv
Tel Aviv University
and
Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

Instruction Level Parallelism (ILP)

Architectures capable of simultaneous execution of multiple instructions

- issued at the same time (multiple-issue architectures) or
- issued while preceding instructions still execute (pipelined architecture)
- combination possible: multiple-issue architecture with pipelined functional units

Main distinction between multiple-issue architectures: Who decides when to issue an instruction:

- Compiler statically schedules: VLIW
- Hardware dynamically issues: Superscalar
Structure

1. **Architectural classification**
2. the **scheduling problem** and **dependence**
3. data **dependences in basic blocks**
4. basic-block scheduling for a **simple pipeline**
5. list scheduling for **basic blocks and complex architectures**
6. scheduling for **acyclic sequences of basic blocks**
7. software pipelining for **loops**
The VLIW Architecture

- Several functional units, ideally homogeneous, in practice not,
- One instruction stream, in each instruction at most 1 operation per FU,
- Jump priority rule for several conditional jumps in 1 instruction,
- FUs connected to register banks, otherwise too many ports required.
Pipelining as Architectural Principle

- split operation into a sequence of phases/stages of roughly same duration;
- execute several consecutive instances in an overlapped fashion.
- Principle can be applied to the execution of instructions as well as to the execution of operations in functional units.
Instruction Pipeline

Several instructions in different stages of execution
Potential structure:

1. instruction fetch and decode,
2. operand fetch,
3. instruction execution,
4. write back of the result into target register.
Pipeline hazards

- **Data hazards**: Needed operand not yet available, cf. true dependence
- **Structural hazards**: Resource conflicts, several instructions need same machine resource, e.g. functional unit, bus,
- **Control hazards**: (Conditional) jumps, condition not yet evaluated.
Phases in dynamically scheduled execution

Assuming a load/store architecture.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. fetch &amp; decode</td>
<td>instruction</td>
</tr>
</tbody>
</table>
# Phases in dynamically scheduled execution

<table>
<thead>
<tr>
<th>Phase</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. fetch &amp; decode instruction</td>
<td>detection of structural hazards</td>
</tr>
</tbody>
</table>
## Phases in dynamically scheduled execution

<table>
<thead>
<tr>
<th>Phase</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. fetch &amp; decode</td>
<td>detection of structural hazards</td>
</tr>
<tr>
<td>instruction</td>
<td></td>
</tr>
<tr>
<td>operand</td>
<td></td>
</tr>
</tbody>
</table>
Phases in dynamically scheduled execution

<table>
<thead>
<tr>
<th>Phase</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. fetch &amp; decode instruction</td>
<td>detection of structural hazards</td>
</tr>
<tr>
<td>operand</td>
<td>detection of data hazards</td>
</tr>
</tbody>
</table>
Phases in dynamically scheduled execution

<table>
<thead>
<tr>
<th>Phase</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. fetch &amp; decode instruction</td>
<td>detection of structural hazards</td>
</tr>
<tr>
<td>operand</td>
<td>detection of data hazards</td>
</tr>
<tr>
<td>2. register operand fetch</td>
<td></td>
</tr>
</tbody>
</table>
Phases in dynamically scheduled execution

<table>
<thead>
<tr>
<th>Phase</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. fetch &amp; decode instruction</td>
<td>detection of structural hazards</td>
</tr>
<tr>
<td>operand</td>
<td>detection of data hazards</td>
</tr>
<tr>
<td>2. register operand fetch</td>
<td>dispatch to functional unit</td>
</tr>
</tbody>
</table>
Phases in dynamically scheduled execution

<table>
<thead>
<tr>
<th>Phase</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. fetch &amp; decode instruction</td>
<td>detection of structural hazards</td>
</tr>
<tr>
<td>operand</td>
<td>detection of data hazards</td>
</tr>
<tr>
<td>2. register operand fetch</td>
<td>dispatch to functional unit</td>
</tr>
<tr>
<td>3. execute</td>
<td>execute operation or load/store</td>
</tr>
</tbody>
</table>
## Phases in dynamically scheduled execution

<table>
<thead>
<tr>
<th>Phase</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. fetch &amp; decode instruction</td>
<td>detection of structural hazards</td>
</tr>
<tr>
<td></td>
<td>operand</td>
</tr>
<tr>
<td></td>
<td>detection of data hazards</td>
</tr>
<tr>
<td>2. register operand fetch</td>
<td>dispatch to functional unit</td>
</tr>
<tr>
<td>3. execute</td>
<td>execute operation or load/store</td>
</tr>
<tr>
<td>4. write back</td>
<td>write to register (or store)</td>
</tr>
</tbody>
</table>
Exploiting Parallelism — The Setting

Hardware offers parallel execution,

Code Selector produced a sequential instruction stream,

Goal Discover inherent parallelism in the sequential program,

Question: When?

Exploitable Parallelism based on notion of independence.
Power PC Pipeline
ColdFire Pipeline
Static and Dynamic Scheduling

<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global</strong> dependence analysis:</td>
<td>in each scheduling step:</td>
</tr>
<tr>
<td>in each scheduling step:</td>
<td>with <strong>local</strong> dependence analysis,</td>
</tr>
<tr>
<td>check non-dep. of candidates</td>
<td>check non-dep. of candidates</td>
</tr>
<tr>
<td>on prev. scheduled instructions;</td>
<td>on curr. executing or delayed instructions;</td>
</tr>
<tr>
<td>schedule non-dep. instructions</td>
<td>dispatch or delay non-dep. instructions.</td>
</tr>
<tr>
<td>after appropriate delay</td>
<td></td>
</tr>
<tr>
<td><strong>Scope</strong> can be:</td>
<td><strong>Scope</strong> is a small <strong>Window</strong>,</td>
</tr>
<tr>
<td>Basic Block,</td>
<td>6 - 12 instructions.</td>
</tr>
<tr>
<td>Sequence of basic blocks,</td>
<td>support by scheduling helpful</td>
</tr>
<tr>
<td>Loops.</td>
<td></td>
</tr>
</tbody>
</table>
Structure

1. Architectural classification
2. the scheduling problem and dependence
3. data dependences in basic blocks
4. basic-block scheduling for a simple pipeline
5. list scheduling for basic blocks and complex architectures
6. scheduling for acyclic sequences of basic blocks
7. software pipelining for loops
Instruction Scheduling

- Reorders instruction stream as generated by instruction selection,
- Goal: Exploitation of intraprocessor parallelism,
  - Filling very long instruction words (VLIWs), or
  - Avoiding pipeline hazards.
- Must be semantics preserving,
- Basis: Program dependences.
Program Dependences

Dependence constrains the potential for reordering:
$S_2$ depends on $S_1 \implies S_1$ must be executed before $S_2$.
$S_1$, $S_2$ can be operations, instructions, basic blocks.
Two types of dependences:

Data Dependence:

- Relation between definitions and uses of resources (program variables, memory cells or blocks, symbolic or real registers),
- Here mainly machine resources, i.e. registers, memory cells, status words
- Alias problems:
  - Address calculation for an index expression
  - Dereferencing of a pointer

Control Dependence: Conditions dominating statements
Example

S1: read a
S2: if a > 0
S3: then b := a;
S4: c := b + a
S5: else c := -(a + a);
S6: d := 2 * b;

S2 is data dependent on S1 — it uses the value computed by S1. S3, S4, S5 are control dependent on S2 — they are only executed depending on the outcome of the test.
Definitions and Uses of Machine Resources

Definitions:

- modifications of register contents by loads or operations, pre-, postincrement/decrement,
- setting carry, overflow, condition bits in status words,
- storing values in memory cells,
- modifying registers as side effects of e.g. pop, push.

Uses:

- Using register contents in operations and for addressing,
- Storing register contents,
- Loading contents of memory cells,
- Testing the program status word.
Types of Data Dependences

Definitions ($X :=$) and uses ($:= X$) of resource $X$:

- $a : X :=$
- $b : X :=$
- $c : := X$
- $d : X :=$

Output dependences (dd, WAW): Definitions on definitions,
e.g., $b$ on $a$,

True dependence (du, RAW): Uses on definitions,
e.g., $c$ on $b$,

Antidependence (ud, WAR): Definitions on uses,
e.g., $d$ on $c$. 
Structure

1. Architectural classification
2. the scheduling problem and dependence
3. data dependences in basic blocks
4. basic-block scheduling for a simple pipeline
5. list scheduling for basic blocks and complex architectures
6. scheduling for acyclic sequences of basic blocks
7. software pipelining for loops
Data Dependence Graph (DDG) (for a basic block)

Nodes instructions,

Edges

- \( a \) sets a resource, \( b \) uses it,
  and the path from \( a \) to \( b \) is definition free, or
- \( a \) uses a resource, \( b \) sets it,
  and the path from \( a \) to \( b \) is definition free, or
- \( a \) and \( b \) set the same resource
  and the path from \( a \) to \( b \) is use and definition free

- describes the degree of freedom for semantics-preserving reordering of the instructions.

\[
\begin{align*}
\text{a: } & X := \\
& \quad \text{definition free} \\
\text{b: } & := X \\
\text{:= } & X \\
& \quad \text{definition free} \\
\text{:= } & X \\
& \quad \text{definition and use free} \\
\end{align*}
\]
Example

DDG

- contains all direct dependencies as edges,
- dependence is transitive, but does not need to be represented,
- transitive closure is an upper approximation due to aliasing,
- direct dependences are enough to prevent non-semantic preserving reorderings.

Instruction sequence with its DDG

1: (CC, D1) := M[A1 + 4].W
2: (CC, D2) := M[A1 + 6].W
3: (CC, D1) := D1 + D2
Eliminating non-live dependences

Flags in the **condition code/program status word**

- are machine resources,
- on some machine set in each arithmetic instruction,
- used in conditional branches.
- Dependences would prevent any reordering due to dd-dependences,
- should be eliminated as shown in figure.
Basic Block with DDG

1: D1 := M[A1+4];
2: D2 := M[A1+6];
3: A1 := A1+2;
4: D1 := D1+A1;
6: D2 := D2+1;
7: D3 := M[A1+12];
8: D3 := D3+D1;
Basic Block with DDG

1: \( D1 := M[A1+4] \);  
2: \( D2 := M[A1+6] \);  
3: \( A1 := A1+2 \);  
4: \( D1 := D1+A1 \);  
6: \( D2 := D2+1 \);  
7: \( D3 := M[A1+12] \);  
8: \( D3 := D3+D1 \);  
Algorithm DDG-Graph

**Input:** basic block
**Output:** data dependence graph of basic block
**Method:** backwards traversal

```plaintext
var firstDefs, expUses: set of pair (resource, instrOcc);
actInstr: instruction;

function conflict(res, instr1, instr2): conflictTyp;
    (* determ. exist. and type of conflict betw. instr1 and instr2 on resource res *)
    if res is set in instr1 then
        if res is used in instr2 then conflictTyp := def-use
            else conflictTyp := def-def fi
    else if res is used in instr1 and set in instr2 then conflictTyp := use-def fi
fi;

procedure drawEdge(a -> b, conflictTyp)
draws a new edge between its arguments if there is none.
```
begin
actInstr := last instruction of basic block;
firstDefs := \{(r, actInstr)\mid r \in \text{def}(actInstr)\};
expUses := \{(r, actInstr)\mid r \in \text{use}(actInstr)\};
while \text{pred}(actInstr) \text{ defined} do
Invariant:
    firstDefs = \{(r, i)\mid i \text{ contains first def. of } r \text{ in } actInstr; \beta\}
    expUses = \{(r, i)\mid i \text{ contains use of } r \text{ not preceded by a def. of } r \}
actInstr := \text{pred}(actInstr);

foreach resource \( r \) set or used in \( actInstr \) do
    foreach \((r, b)\) \in firstDefs \cup expUses do
        case conflict\((r, actInstr, b)\) is
            def-def: if exists no pair \((r, .)\) in expUses
                then drawEdge\((actInstr \rightarrow b, dd)\) fi;
            def-use: drawEdge\((actInstr \rightarrow b, du)\);
            use-def: drawEdge\((actInstr \rightarrow b, ud)\);
        end case
    od
od;
(* Updating firstDefs and expUses *)

foreach resource $r'$ set in actInstr do
  firstDefs := firstDefs - $\{(r', .) \in \text{firstDefs}\} \cup \{(r', \text{actInstr})\}$;
  expUses := expUses - $\{(r', .) \in \text{expUses}\}$
  od;

foreach resource $r'$ used in actInstr do
  expUses := expUses $\cup \{(r', \text{actInstr})\}$;
  od
Invariant restored!

od
end
Structure

1. Architectural classification
2. the scheduling problem and dependence
3. data dependences in basic blocks
4. basic-block scheduling for a simple pipeline
5. list scheduling for basic blocks and complex architectures
6. scheduling for acyclic sequences of basic blocks
7. software pipelining for loops
(Simple-) Pipeline Scheduling

Simple pipeline with the following properties:

- instruction pipeline without hazard detection, i.e., no pipeline interlocks,
- simple resource model: instruction uses 1 resource for 1 cycle \(\implies\) different instructions scheduled on different cycles do not interfere,
- one cycle delay for true-dependent instructions,
- goal: hiding latencies to minimize program length.

Later, complex resource models: instruction occupies a resource for more than 1 cycle.
Complexity and Heuristics

- **Optimal Pipeline Scheduling**, even for simple pipelines, is an NP-complete problem,
- use **topological sorting** to convert partial order into total order
- In the example, several possible linear order exist, e.g.
  \{1, 2, 6, 3, 5, 9, 4, 7, 8, 10, 11\},
  \{6, 5, 1, 2, 3, 7, 4, 10, 11, 9, 8\}
- use heuristics for the selection of candidates next to be scheduled:
  ▶ candidates with most dependences,
  ▶ candidates on the longest path.
Algorithm Pipeline Scheduling

(Gibbons/Muchnick 1986)

**Input**: Basic block with DDG, set of schedules for preceding basic blocks.

**Output**: (Possibly) reordered instruction sequence of the basic block, possibly with inserted NOPs.

**Method**: topol. sorting constrained by the pipeline conditions

```plaintext
var cands, realCands, potColls: set of instrOcc;

(* cands: instructions without predecessor *)
(* potColls: already scheduled instructions whose delay is not over *)
(* realCands: instructions in cands without conflict with potColls *)

function colliding(cand, potCol)/ : set of instrOcc;
    computes the set of instructions in cand, colliding with those in potColls
```
begin
   \text{cands} := \text{set of minimal elements of the DDG};
   \text{potColls} := \text{set of last instructions in schedules of preceding basic block};
repeat
   \text{realCands} := \text{cands} - \text{colliding}(\text{cands}, \text{potColls});
   \text{if } \text{realCands} \neq \emptyset \text{ then}
      \begin{align*}
      & \text{evaluate candidates according to heuristics;} \\
      & \text{select a best candidate } b; \quad \text{schedule } b; \\
      & \text{remove } b \text{ from } \text{cands}; \\
      & \text{remove } b \text{ and all outgoing edges from the DDG}; \\
      & \text{insert new minimal elements into } \text{cands}; \\
      & \text{potColls} := \{ b \}
      \end{align*}
   \text{else schedule a NOP; } \text{potColls} := \emptyset
fi
until \text{cands} = \emptyset
end
Example
Structure

1. Architectural classification
2. the scheduling problem and dependence
3. data dependences in basic blocks
4. basic-block scheduling for a simple pipeline
5. list scheduling for basic blocks and complex architectures
6. scheduling for acyclic sequences of basic blocks
7. software pipelining for loops
More Complex Architectures

- Parallel functional units,
- Complex resource patterns — multi-cycle operations,

Require modifications of algorithm **Pipeline Scheduling**

- uses resource usage patterns for instructions and resource constraints for the architecture,
- may schedule several instructions in the same position,
- keeps list of data-ready instructions, i.e., instructions whose (dependence) predecessors will have produced their results in time for the current instruction,
- chooses from the ready list by a priority heuristics,
- keeps a global resource table for bookkeeping about occupied resources and for checking for resource conflicts.
More Complex Architecture — New Terminology

**Operation**: Machine Operation, e.g. **Load**, **Store**, **Add**
*generic names*: $a, b, c, \ldots$

**Instruction**: Set of operations scheduled at the same position,
*generic names*: $A, B, C, \ldots$

**Latency**: Execution time of an operation

**Delay**: Required distance between the issue of $a$ and the
issue of $b$ if $(a \rightarrow b)$

**Schedule**: Mapping from operations to positions (cycles),
*generic names*: $\sigma, \sigma_{flat}, \sigma_{swap}, \ldots$
Delays as Functions of Dependence Type

Delay for \((a \rightarrow^{dt} b)\) depends on the latencies of \(a\) and \(b\) and \(dt\). Assumptions:

- **write**-cycle is the last,
- **read**-cycles is any cycle but the last,
- in concurrent **reads** and **writes**, **read** reads old content.

\[
du: \quad \text{latency}(a)
\]
\[
ud: \quad -1 + \text{latency}(a) - \text{latency}(b)
\]
\[
dd: \quad 1 + \text{latency}(a) - \text{latency}(b)
\]
Algorithm List Scheduling

**Input**: Basic block with DDG, set of schedules for preceding basic blocks.

**Output**: Instruction sequence of the basic block associated with times (positions in the schedule).

**Method**: topological sorting constrained by the pipeline conditions

```plaintext
var time: int;
vary cands: set of instrOcc;
array GRT[R × ...] of Bool;
(* GRT[r, t] = true iff constructed schedule occupies resource r at time t *)
function resConflict(cand, grt) : bool;
  checks whether cand has a resource conflict with the current schedule;
```
The Global Reservation Table, GRT
begin
  \( time := 0; \text{cands} := \text{set of minimal elements of the DDG}; \)
  \textbf{while} \text{cands} \neq \emptyset \textbf{then}
    \text{sort cands in non-decreasing priority order;}
  \textbf{while} \text{not all candidates have been tried} \textbf{do}
    \text{check next candidate b for resource conflicts;}
    \textbf{if} \text{not resConflict}(b, GRT) \textbf{then} \text{schedule b at time;}
    \text{update GRT;}
    \text{remove b from cands;}
    \text{remove b and all outgoing edges from the DDG;}
  \textbf{od}
  \textbf{od}
  \text{increment time by 1; update cands;}
  \text{insert instructions whose delay is over into cands;}
end
Structure

1. Architectural classification
2. the scheduling problem and dependence
3. data dependences in basic blocks
4. basic-block scheduling for a simple pipeline
5. list scheduling for basic blocks and complex architectures
6. scheduling for acyclic sequences of basic blocks
7. software pipelining for loops
Exposing more Instruction Level Parallelism

Degree of ILP in basic blocks is limited – typically to 2
Available ILP in processors grows: better exploitation by
  ▶ Scheduling sequences of consecutive basic blocks
  ▶ Scheduling loops
  ▶ Speculation

<table>
<thead>
<tr>
<th>when</th>
<th>what</th>
<th>how to preserve the semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>dynamic</td>
<td>hardware branch prediction</td>
<td>on a mispredicted branch – forgetting or undoing effects of speculatively executed instructions</td>
</tr>
<tr>
<td>static</td>
<td>speculative code motion</td>
<td>compensation code</td>
</tr>
</tbody>
</table>
Code Motion

- moves code from a source block to a target block,
- upward code motion: target block is predecessor of source block,
- downward code motion: target block is successor of source block,
- code motion is speculative if the moved code is executed on some control-flow path on which it would not have been executed before.
- code motion may require the insertion of duplicates (compensation code), if some moved code were not executed on some control path.
Trace-/ Super- / Hyperblock Scheduling

What is the total running time of a program?

\[ \sum_{basic\ block\ i} t_i \times f_i \]

where \( t_i \) is the duration and \( f_i \) the frequency of execution of basic block \( i \).

Do we know \( t_i \) and \( f_i \)? — In general, we don’t! Profiling computes an approximation to them.
Trace- / Superblock- / Hyperblock-Scheduling

- Extend scheduling area to sequences of consecutive basic blocks (*traces, superblocks, hyperblocks*),
- Select frequently taken paths based on profile data, annotate program with profiling information: associate each branch of a conditional with a relative frequency,
- Optimize and schedule frequently taken traces at the cost of less frequently taken traces.
Traces

**Trace** is a sequence of consecutive basic blocks not extending across a loop boundary.
Control flow graph of a procedure is partitioned into a disjoint set of traces:

- traces formed in order of decreasing frequency:
  1. select available basic block with highest frequency
  2. join available predecessors and successors with highest frequencies until frequency falls below a given threshold

- there are (unlike in basic blocks)
  - side exits out of traces
  - side entrances into traces
A Partitioning into Traces
Example

for i := 0 upto n do
    if A[i] = 0
        then B[i] := B[i] + s
    else B[i] := A[i]
    fi
    sum := sum + B[i]
od

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
|   | R1 := 0 (* stepping thru A and B *)
| i1 | R5 := 0 (* holds sum *)
| i2 | R6 := n
| i3 | R7 := s
| i4 | R2 := M[ R1 + a ]
| i5 | BNE R2 0 i7
| i6 | R3 := M[ R1 + b ]
| i7 | R4 := R3 + R7
| i8 | M[ R1 + b ] := R4
| i9 | BR i9
| i10| R4 := R2
| i11| M[ R1 + b ] := R2
| i12| R5 := R5 + R4
| i13| R1 := R1 + 4
| i14| BLT R1 R6 i1
|   |   |
Basic-Block Schedule for Example

Assumptions: 2-issue processor with 2 integer units, latency of arithmetic and of store is 1 cycle, of load is 2 cycles, no stall cycles for a branch.
Trace Scheduling

List scheduling applied to a trace – Problems:
- code motion past side exits
- code motion past side entrances

may destroy semantics.

Compensation code inserted on off-trace paths.

Problems:
- Code growth
- Exceptions raised by compensation code moved in front of side exits
## Trace Schedule for Example

<table>
<thead>
<tr>
<th>Time</th>
<th>Int. Unit 1</th>
<th>Int. Unit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2</td>
<td>i1 R2 := M[R1 + a]</td>
<td></td>
</tr>
<tr>
<td>3 4 5</td>
<td>i3 R3 := M[R1 + b]</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>i4 R4 := R3 + R7</td>
<td>i6 BR i9</td>
</tr>
<tr>
<td>7 (4)</td>
<td>i9 R5 := R5 + R4</td>
<td>i8 M[R1 + b] := R2</td>
</tr>
<tr>
<td>8 (5)</td>
<td>i11 BLT R1 R6 i1</td>
<td></td>
</tr>
</tbody>
</table>

### Basic-Block Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Int. Unit 1</th>
<th>Int. Unit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i1 R2 := M[R1 + a]</td>
<td>i3 R3 := M[R1 + b]</td>
</tr>
<tr>
<td>1</td>
<td>i2 BNE R2 0 i7</td>
<td>i4 R4 := R3 + R7</td>
</tr>
<tr>
<td>2</td>
<td>i5 M[R1 + b] := R4</td>
<td>i10 R1 := R1 + 4</td>
</tr>
<tr>
<td>3 (5)</td>
<td>i9 R5 := R5 + R4</td>
<td></td>
</tr>
<tr>
<td>4 (6)</td>
<td>i11 BLT R1 R6 i1</td>
<td></td>
</tr>
<tr>
<td>6 (7)</td>
<td>i12 BR out</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>i7 R4 := R2</td>
<td>i8 M[R1 + b] := R2</td>
</tr>
<tr>
<td>8</td>
<td>i12 BR i9</td>
<td></td>
</tr>
</tbody>
</table>

### Trace Schedule
Speculative Upward Code Motion

**Correct**

**Wrong**

Solution: Register renaming
Downward Code Motion with Insertion of Compensation Code

\[ X := \]

- \( s_1 \):
  - if \( e \) then \( s_2 \)
    - \( X \) live at \( s_2 \)

- \( s_1 \):
  - if \( e \) then \( s_2 \)
    - \( X \) not live at \( s_2 \)

\[ \text{dependence} \quad \text{no dependence} \]
Moving a Statement past a Side Entrance

- Upwards move of a statement over a side entrance in (a) and (b).
- Rule for these moves in (c)
Superblocks

Avoiding code motion past side entrances by *tail duplication*: copying code starting with side entrance and redirecting the branches.

Superblock formation
- starts with a trace,
- produces a trace without side entrances,
- only one entry, but potentially several exits.

Compensation code only for downward code motion past side exits.
Enlarging Superblocks to increase the available ILP

Branch Target Expansion: “Expands” the last branch of a superblock by copying and appending the target superblock

Loop Peeling and Unrolling: Unroll several iterations of the loop;
  ▶ remove control transfer if safe
  ▶ extend superblock by predecessors and/or successors if possible

Removal of Dependencies:

  register renaming removes artificial dependencies
  operation migration moves an operation from a superblock which does not use the result to another one which does
  induction variable expansion introduces a new instance of an induction variable for every unrolled iteration of a loop;
  removes dependencies of induction variables on themselves;
  requires initialization and finalization code.