Code Scheduling

– Wilhelm/Maurer: Compiler Design, Chapter 12 –

Mooly Sagiv
Tel Aviv University
and
Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

19. November 2007

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 architeture)
- combination possible: multiple-issue architecture with pipelined functional units

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

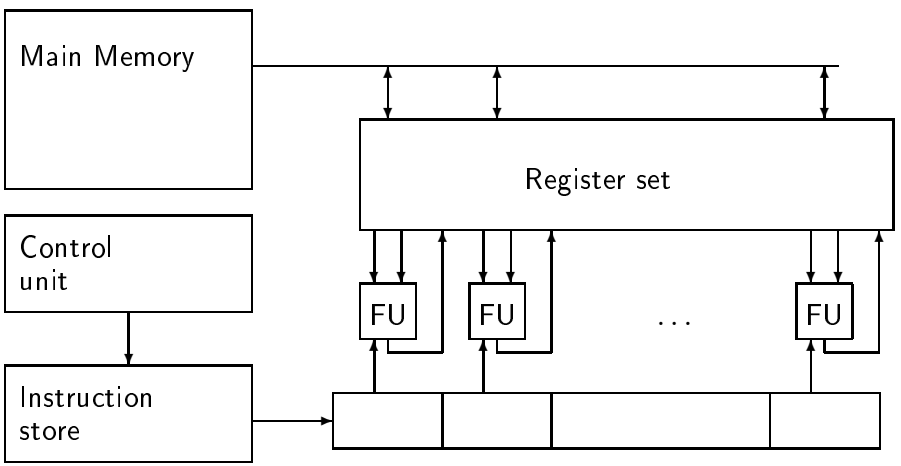
- Compiler statically schedules: VLIW
- Hardware dynamically issues: Superscalar

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.

		cycle						
		1	2	3	4	5	6	7
Pipe-	1	B_1	B_2	B_3	B_4			
line-	2		B_1	B_2	B_3	B_4		
stage	3			B_1	B_2	B_3	B_4	
	4				B_1	B_2	B_3	B_4

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. funcional unit, bus,
- Control hazards: (Conditional) jumps, condition not yet evaluated.

Phases in dynamically scheduled execution

Assuming a load/store architecture.

Phase	Activity
1. fetch & decode instruction	

Phases in dynamically scheduled execution

Phase	Activity
1. fetch & decode instruction	
	detection of structural hazards

Phases in dynamically scheduled execution

Phase	Activity
1. fetch & decode instruction	
	detection of structural hazards
operand	

Phases in dynamically scheduled execution

Phase	Activity
1. fetch & decode instruction	
	detection of structural hazards
operand	
	detection of data hazards

Phases in dynamically scheduled execution

Phase	Activity
1. fetch & decode instruction	
	detection of structural hazards
operand	
	detection of data hazards
2. register operand fetch	

Phases in dynamically scheduled execution

Phase	Activity
1. fetch & decode	
instruction	
	detection of structural hazards
operand	
	detection of data hazards
2. register operand fetch	
	dispatch to functional unit

Phases in dynamically scheduled execution

Phase	Activity
1. fetch & decode	
instruction	
	detection of structural hazards
operand	
	detection of data hazards
2. register operand fetch	
	dispatch to functional unit
3. execute	execute operation or load/store

Phases in dynamically scheduled execution

Phase	Activity
1. fetch & decode	
instruction	
	detection of structural hazards
operand	
	detection of data hazards
2. register operand fetch	
	dispatch to functional unit
3. execute	execute operation or load/store
4. write back	write to register (or store)

Exploiting Parallelism — The Setting

Hardware offers parallel execution,

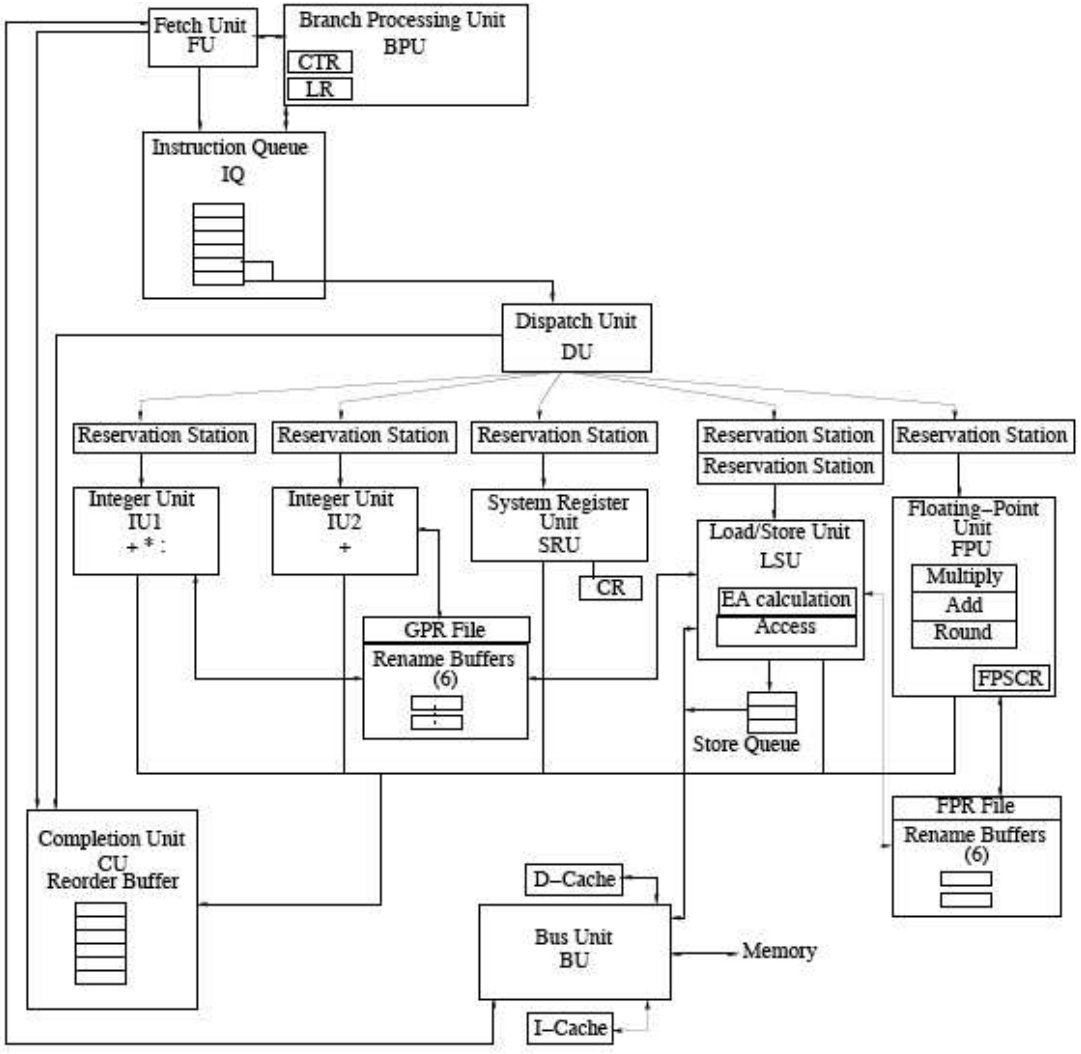
Code Selector produced a sequential instruction stream,

Goal Discover inherent parallelism in the sequential program,

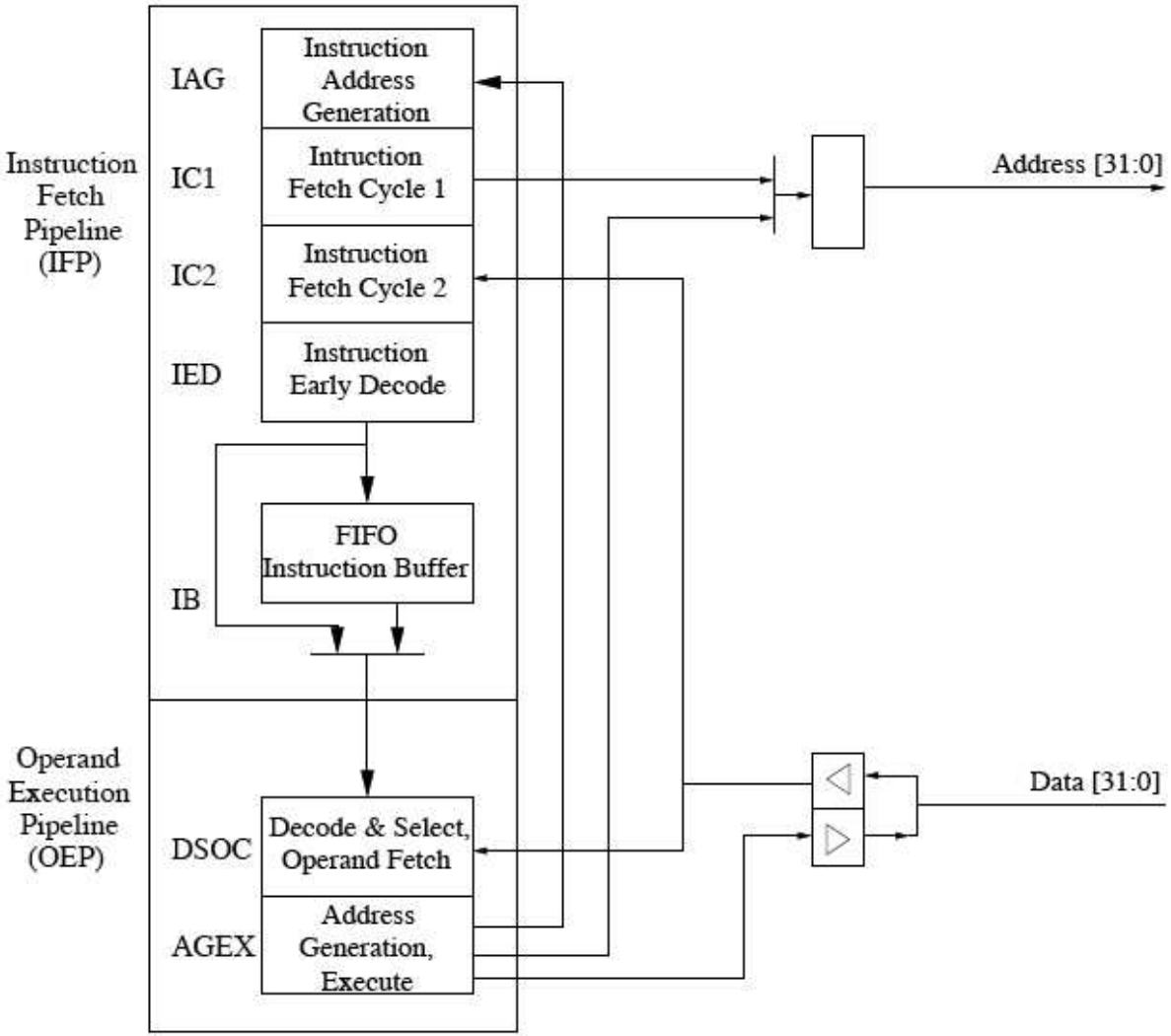
Question: When?

Exploitable Parallelism based on notion of independence.

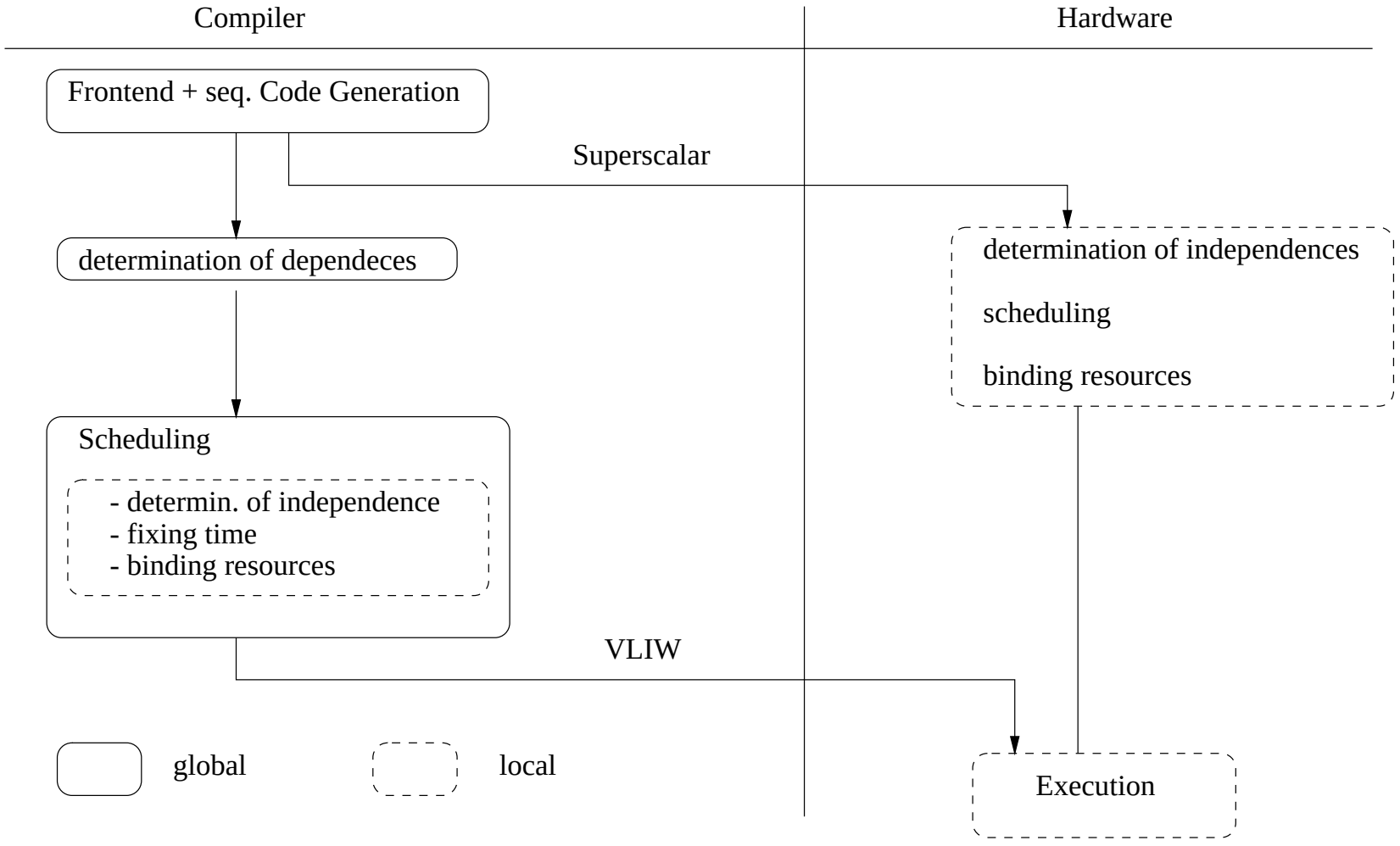
Power PC Pipeline



ColdFire Pipeline



Architecture Characterization



Static and Dynamic Scheduling

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

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 $S_1 \Longrightarrow S_1$ must be executed before S_2.
S_1, S_2 can be operations, instructions, basic blocks.
Two types of dependences:

Data Dependence:

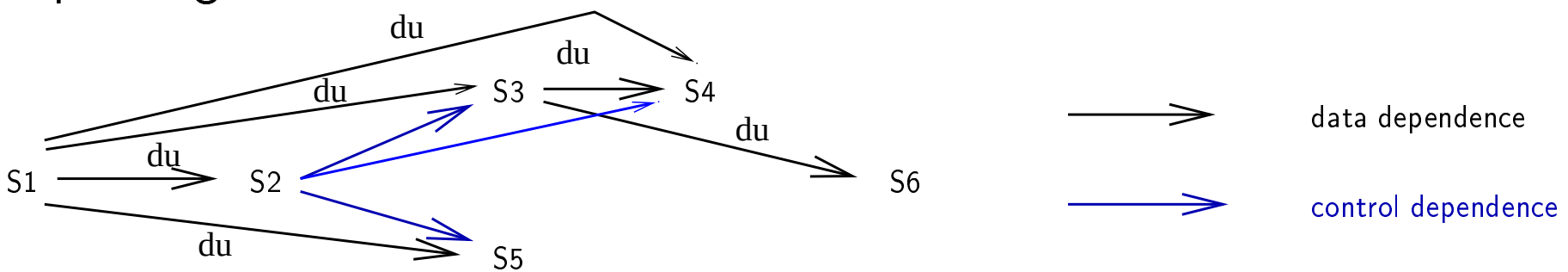
- Relation between definitions and uses of resources (program variables, memory cells or blocks, symbolic or real registers),
- Here mainly machine resources, i.e. registers, memory cells, status words
- Alias problems:
  - Address calculation for an index expression
  - Dereferencing of a pointer

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

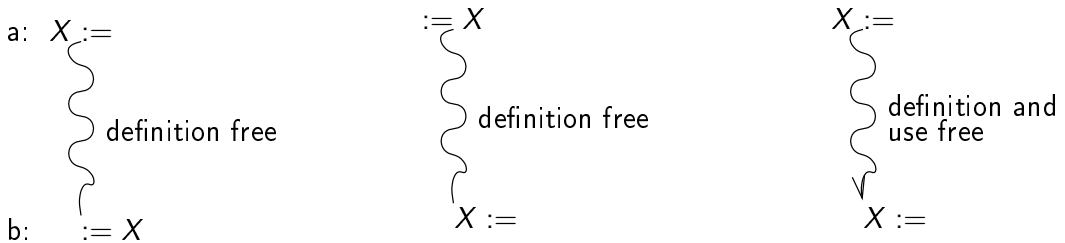
Data Dependence Graph (DDG) (for a basic block)

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.



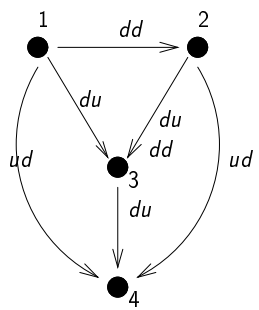
Example

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

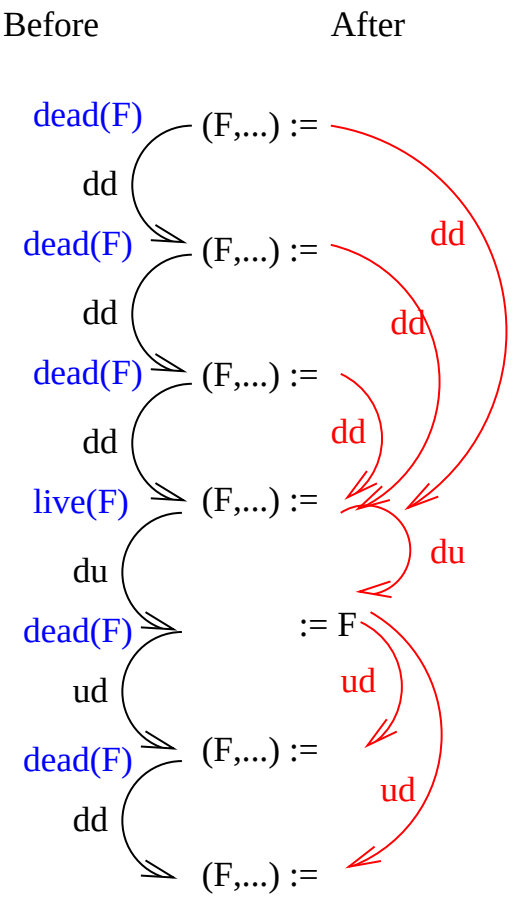
$$
\begin{array}{lll}
1: & (CC, D1) & := M[A1 + 4].W \\
2: & (CC, D2) & := M[A1 + 6].W \\
3: & (CC, D1) & := D1 + D2 \\
4: & M[A1 + 4] & := D1.W
\end{array}
$$



Eliminating non-live dependences

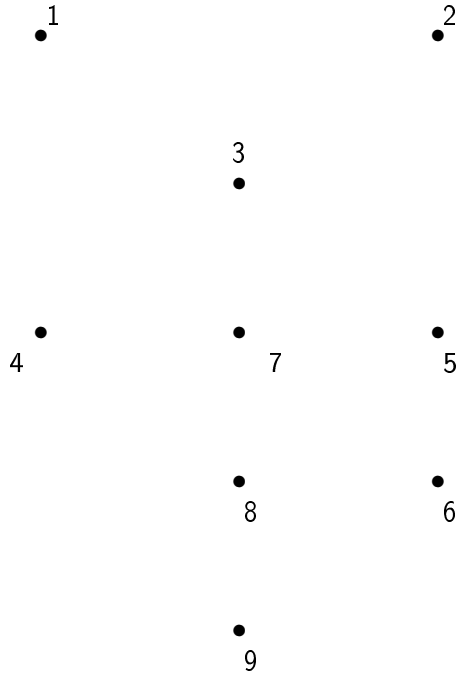
Flags in the condition code/program status word

- 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.



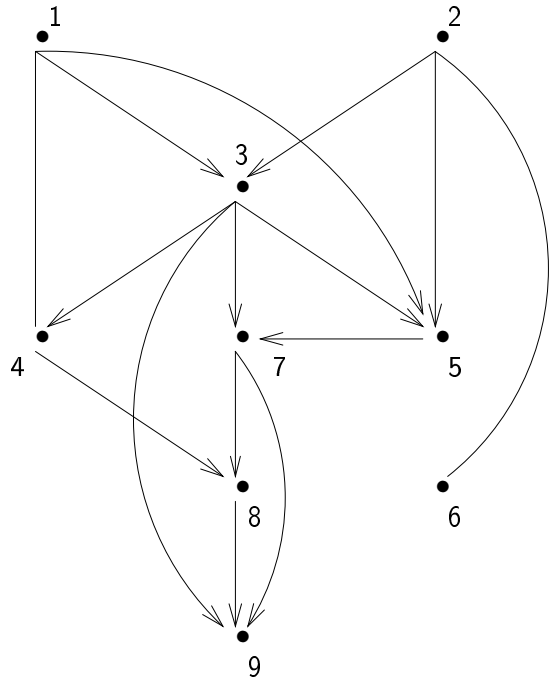
Basic Block with DDG

```
1:  D1      := M[A1+4];
2:  D2      := M[A1+6];
3:  A1      := A1+2;
4:  D1      := D1+A1;
5:  M[A1]   := A1;
6:  D2      := D2+1;
7:  D3      := M[A1+12];
8:  D3      := D3+D1;
9:  M[A1+6] := D3
```



Basic Block with DDG

```
1:  D1      := M[A1+4];
2:  D2      := M[A1+6];
3:  A1      := A1+2;
4:  D1      := D1+A1;
5:  M[A1]   := A1;
6:  D2      := D2+1;
7:  D3      := M[A1+12];
8:  D3      := D3+D1;
9:  M[A1+6] := D3
```



Algorithm DDG-Graph

Input: basic block
Output: data dependence graph of basic block
Method: backwards traversal
var *firstDefs*, *expUses*: **set of pair** (*resource*, *instrOcc*);
  actInstr: *instruction*;

function *conflict*(*res*,$instr_1$,$instr_2$): *conflictTyp*;
  (* determ. exist. and type of conflict betw. $instr_1$ and $instr_2$ on resource *res* *
  if *res* is set in $instr_1$ **then**
  if *res* is used in $instr_2$ **then** *conflictTyp* := *def-use*
    else *conflictTyp* := *def-def* **fi**
  else if *res* is used in $instr_1$ and set in $instr_2$ **then** *conflictTyp* := *use-def* **fi**
fi;
procedure *drawEdge*($a \rightarrow b$, *conflictTyp*)
  draws a new edge between its arguments if there is none.

begin
actInstr := last instruction of basic block;
firstDefs := $\{(r, actInstr) | r \in defs(actInstr)\}$;
expUses := $\{(r, actInstr) | r \in uses(actInstr)\}$;
while *pred*(*actInstr*) *defined* **do**
Invariant:
firstDefs = $\{(r, i) | i$ contains first def. of r in *actInstr*; $\beta\}$
expUses = $\{(r, i) | i$ contains use of r not preceded by a def. of r $\}$
actInstr := *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**
$firstDefs := firstDefs - \{(r', .) \in firstDefs\} \cup \{(r', actInstr)\}$;
$expUses := expUses - \{(r', .) \in expUses\}$
od;
foreach resource r' used in *actInstr* **do**
$expUses := expUses \cup \{(r', actInstr)\}$;
od
Invariant restored!
od
end

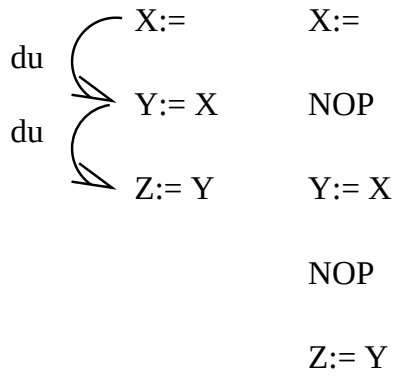
Structure

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:

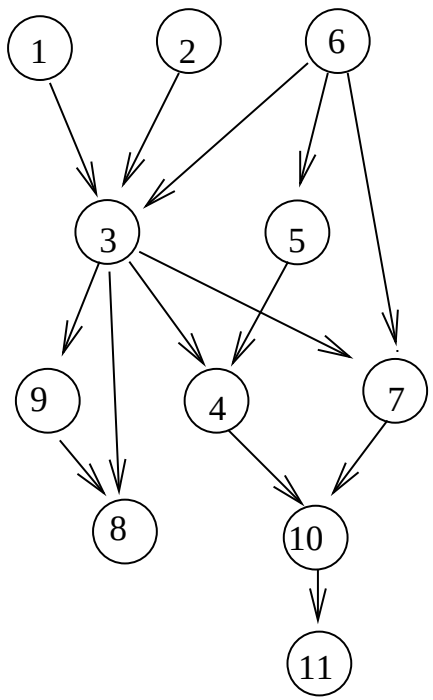
- instruction pipeline without hazard detection, i.e., no pipeline interlocks,
- simple resource model: instruction uses 1 resource for 1 cycle $\Longrightarrow$ different instructions scheduled on different cycles do not interfer,
- one cycle delay for true-dependent instructions,
- goal: hiding latencies 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

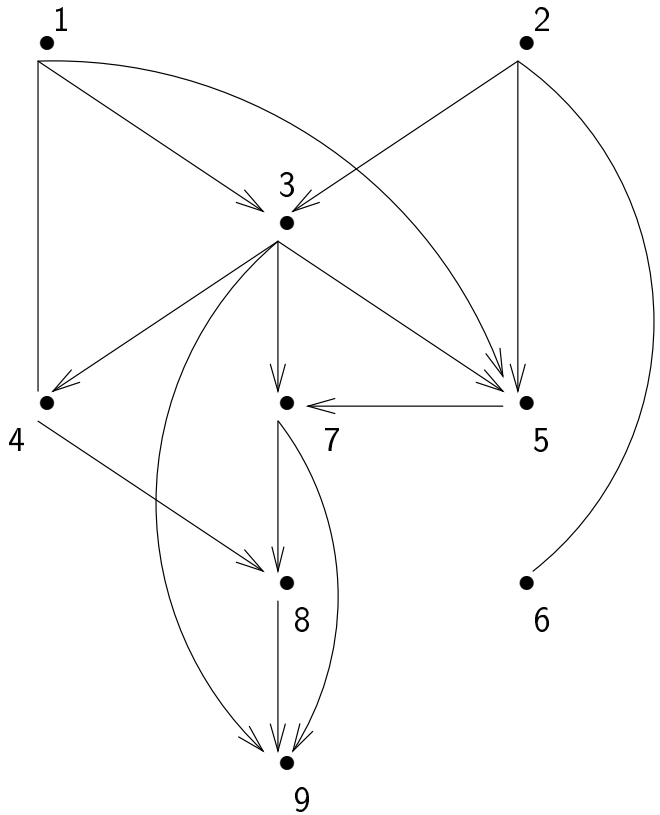
var *cands*, *realCands*, *potColls*: **set of** *instrOcc*;

(* *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
  cands := set of minimal elements of the DDG;
  potColls := set of last instructions in schedules of preceding basic block
  repeat
    realCands := cands − colliding(cands, potColls);
    if realCands ≠ ∅ then
      evaluate candidates according to heuristics;
      select a best candidate b;    schedule b;
      remove b from cands;
      remove b and all outgoing edges from the DDG;
      insert new minimal elements into cands;
      potColls := {b}
    else schedule a NOP; potColls := ∅
    fi
  until cands = ∅
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: $\sigma, \sigma_{flat}, \sigma_{swp}, \ldots$

Delays as Functions of Dependence Type

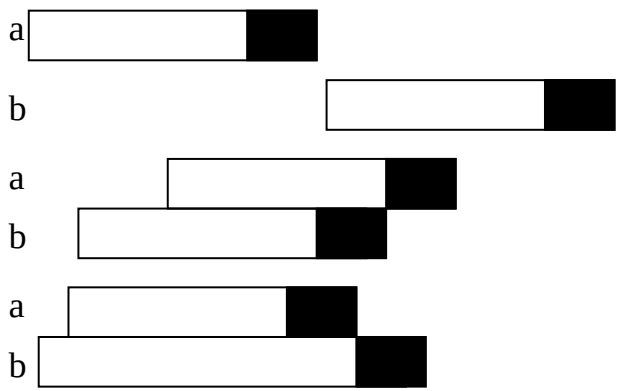
Delay for $(a \rightarrow^{dt} b)$ depends on the latencies of a and b and dt.
Assumptions:

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

du: $latency(a)$

ud: $-1 + latency(a) - latency(b)$

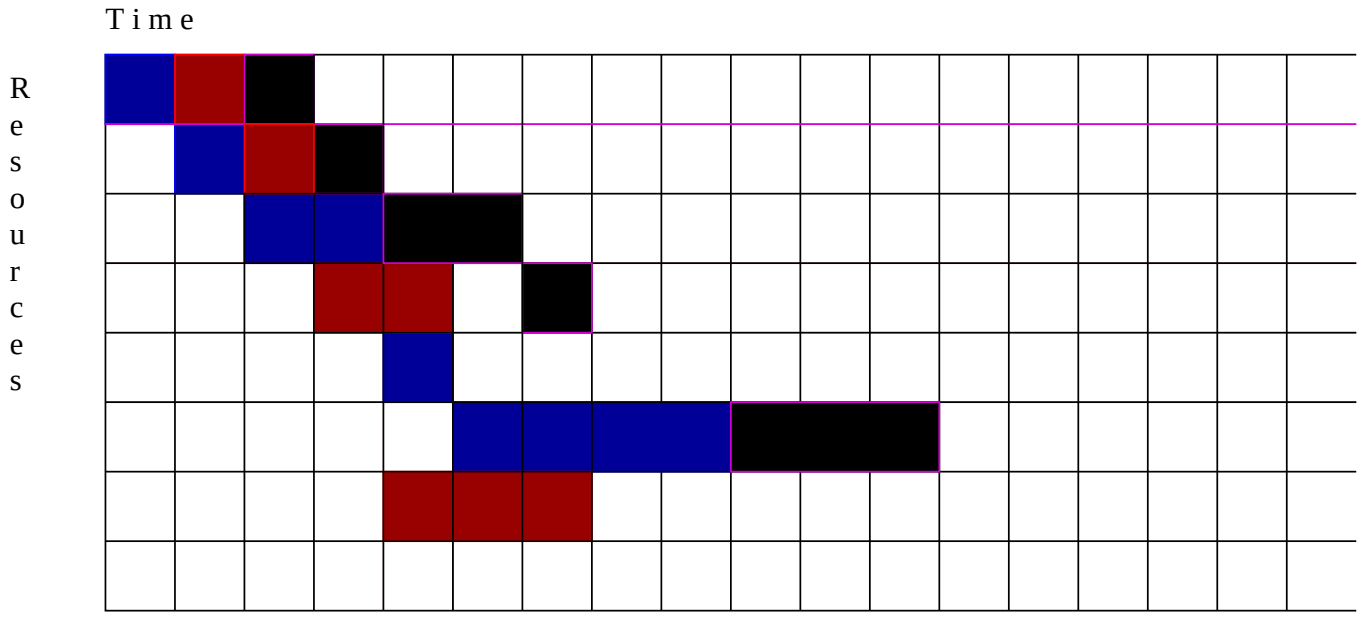
dd: $1 + latency(a) - latency(b)$



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
var *time*: **int**;
var *cands*: **set of** *instrOcc*;
array $GRT[R \times \ldots]$ **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; cands := set of minimal elements of the DDG;
  while cands ≠ ∅ then
    sort cands in non-decreasing priority order;
    while not all candidates have been tried do
      check next candidate b for resource conflicts;
      if not resConflict(b, GRT) then schedule b at time;
      update GRT;
      remove b from cands;
      remove b and all outgoing edges from the DDG;
    od
  od
  increment time by 1; update cands;
  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

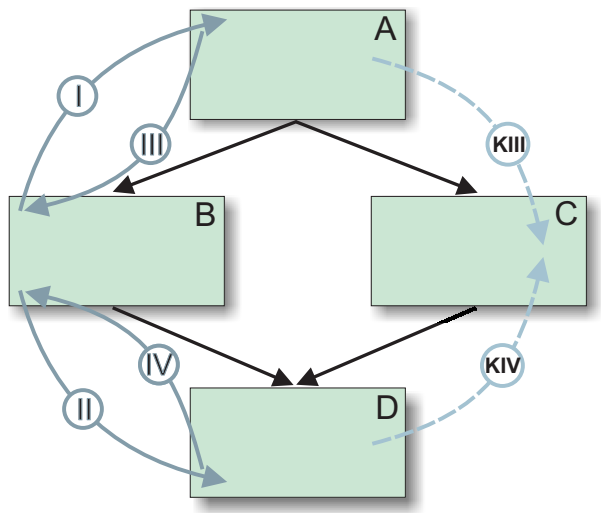
Degree of ILP in basic blocks is limited – typically to 2
Available ILP in processors grows: better exploitation by

- Scheduling sequences of consecutive basic blocks
- Scheduling loops
- Speculation

when	what	how to preserve the semantics
dynamic	hardware branch prediction	on a mispredicted branch – forgetting or undoing effects of speculatively executed instructions
static	speculative code motion	compensation code

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

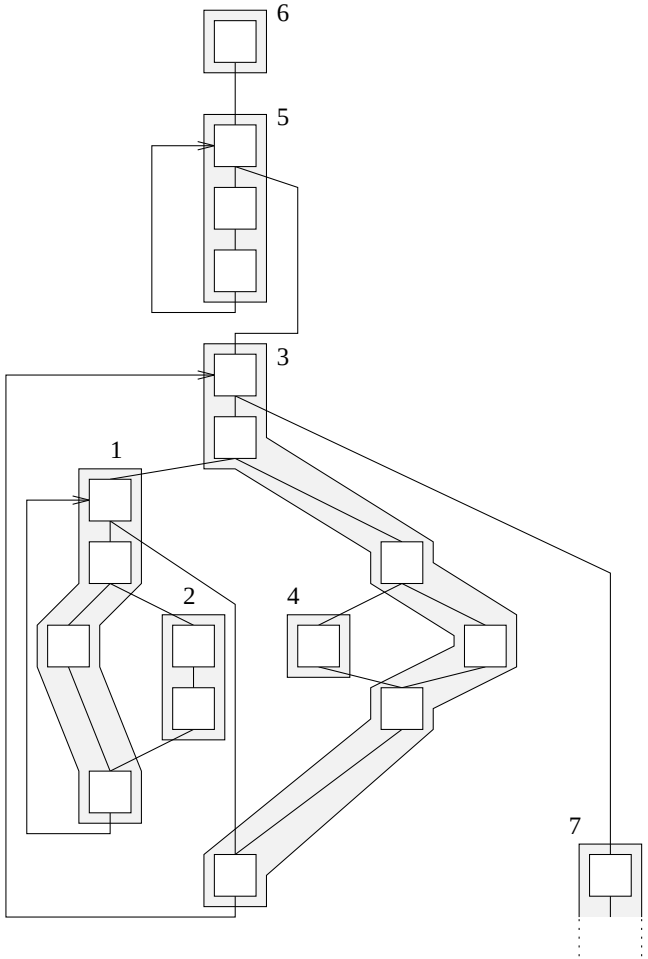
Control flow graph of a procedure is partitioned into a disjoint set of traces

- 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

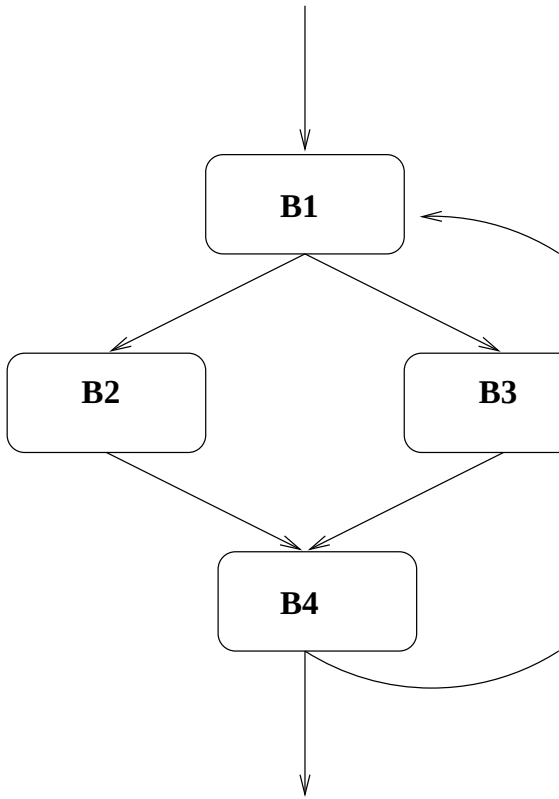
A Partitioning into Traces



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
```

		R1 := 0 (* stepping thru A and B *) R5 := 0 (* holds sum *) R6 := n R7 := s
B1	i1 i2	R2 := M[R1 + a] BNE R2 0 i7
B2	i3 i4 i5 i6	R3 := M[R1 + b] R4 := R3 + R7 M[R1 + b] := R4 BR i9
B3	i7 i8	R4 := R2 M[R1 + b] := R2
B4	i9 i10 i11	R5 := R5 + R4 R1 := R1 + 4 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.

		R1 := 0 (* stepping thru A and B *) R5 := 0 (* holds sum *) R6 := n R7 := s
B1	i1 i2	R2 := M[R1 + a] BNE R2 0 i7
B2	i3 i4 i5 i6	R3 := M[R1 + b] R4 := R3 + R7 M[R1 + b] := R4 BR i9
B3	i7 i8	R4 := R2 M[R1 + b] := R2
B4	i9 i10 i11	R5 := R5 + R4 R1 := R1 + 4 BLT R1 R6 i1

Time	Int. Unit 1		Int. Unit 2	
0	i1	R2 := M[R1 + a]		
1				
2	i2	BNE R2 0 i7		
3	i3	R3 := M[R1 + b]		
4				
5	i4	R4 := R3 + R7		
6	i5	M[R1 + b] := R4	i6	BR i9
3	i7	R4 := R2	i8	M[R1 + b] := R2
7 (4)	i9	R5 := R5 + R4	i10	R1 := R1 + 4
8 (5)	i11	BLT R1 R6 i1		

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

Time	Int. Unit 1		Int. Unit 2	
0	i1	R2 := M[R1 + a]		
1				
2	i2	BNE R2 0 i7		
3	i3	R3 := M[R1 + b]		
4				
5	i4	R4 := R3 + R7		
6	i5	M[R1 + b] := R4	i6	BR i9
3	i7	R4 := R2	i8	M[R1 + b] := R2
7 (4)	i9	R5 := R5 + R4	i10	R1 := R1 + 4
8 (5)	i11	BLT R1 R6 i1		

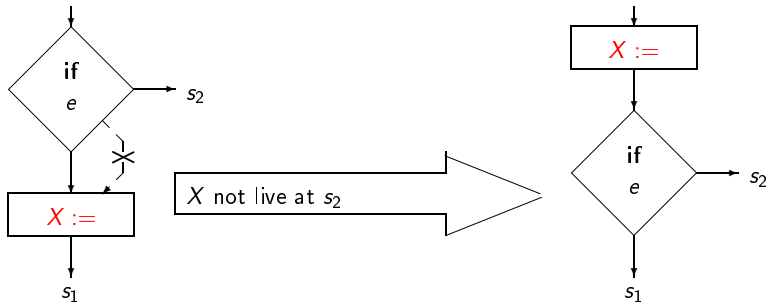
Basic−Block Schedule

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

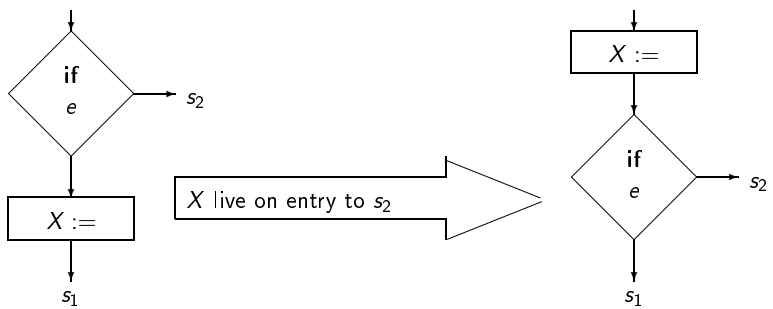
Trace Schedule

Speculative Upward Code Motion

correct

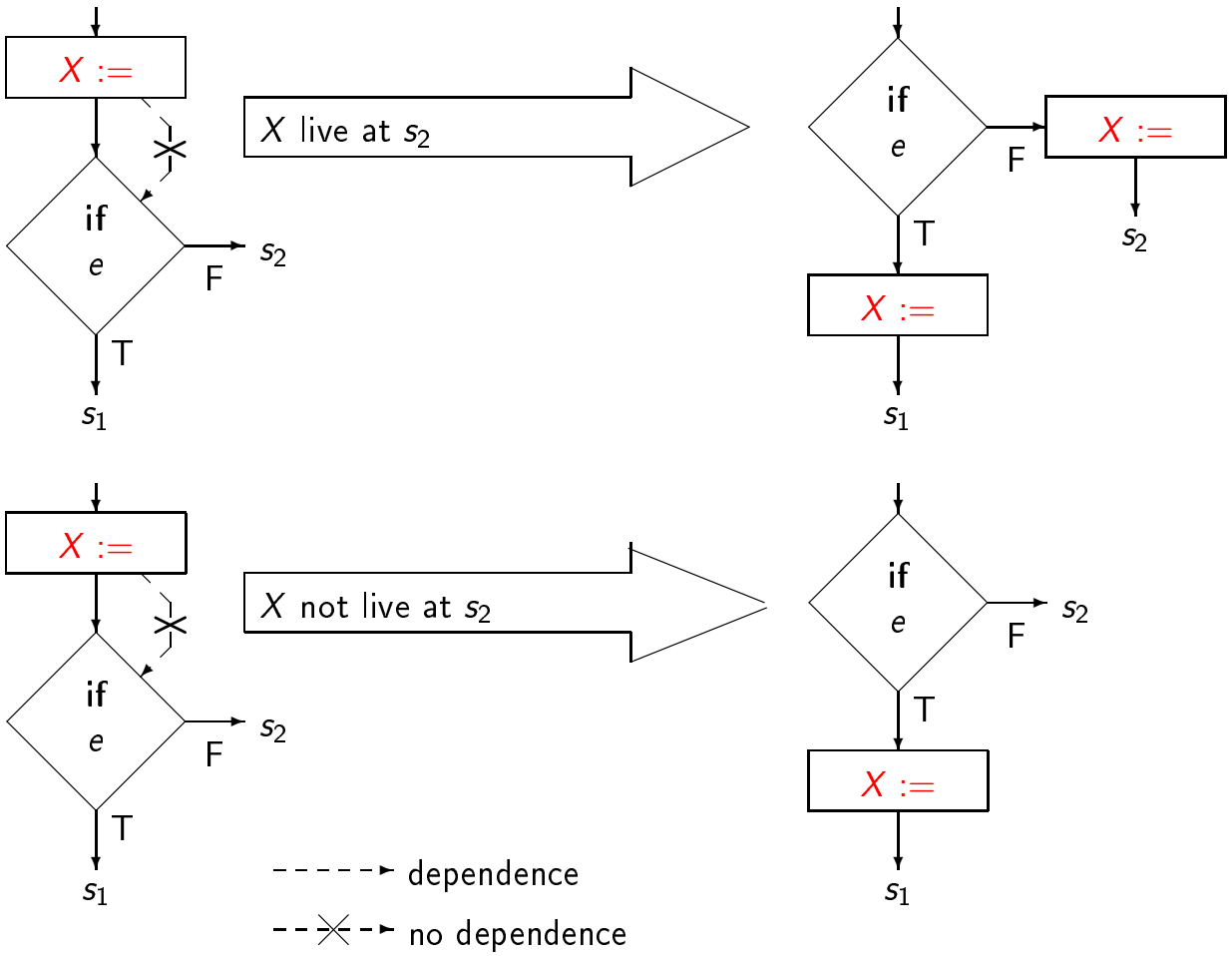


wrong



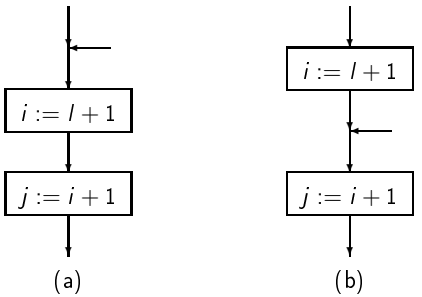
Solution: Register renaming

Downward Code Motion with Insertion of Compensation Code



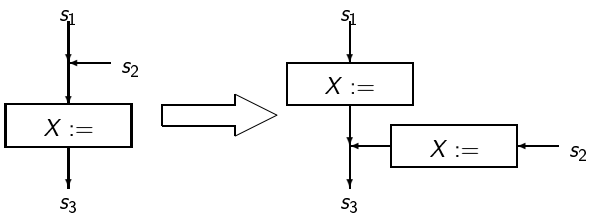
Moving a Statement past a Side Entrance

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



(a)

(b)



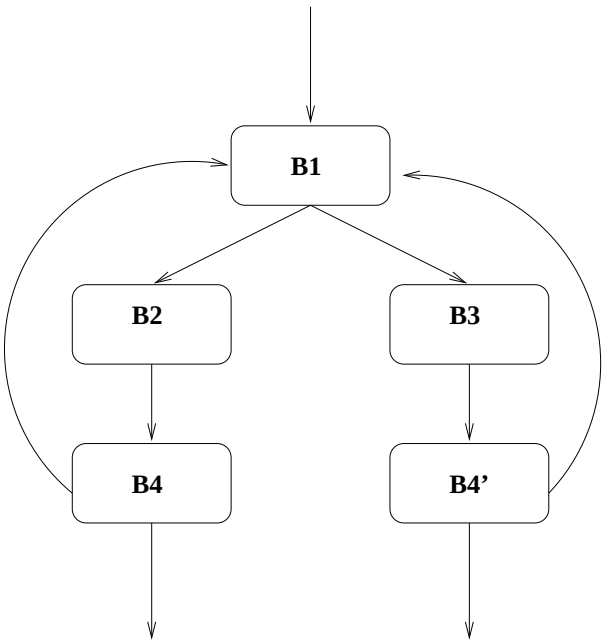
(c)

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.