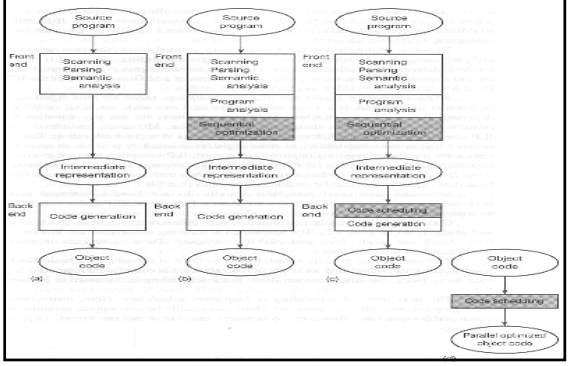
9. Code Scheduling for ILP-Processors

- {Software! compilers optimizing code for ILP-processors, including VLIW}

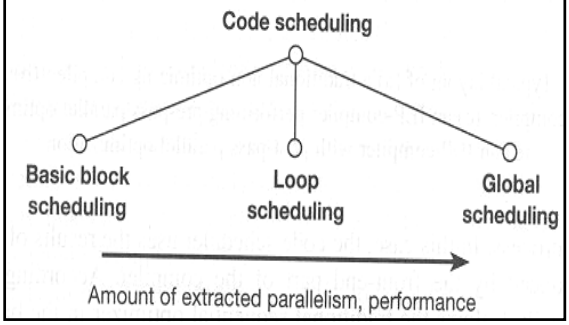
- 9.1 Introduction
- 9.2 Basic block scheduling
- 9.3 Loop scheduling
- 9.4 Global scheduling

TECH
Computer Science

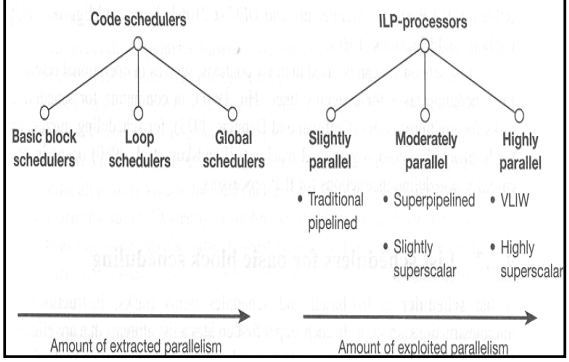
Typical layout of compiler:
traditional, optimizing, pre-pass parallel, post-pass parallel



Levels of static scheduling

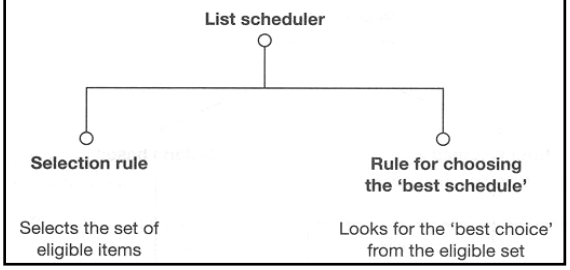


Contrasting the performance of {software/hardware} static code schedulers and ILP-processor



9.1 Basic Block Schedulers: List scheduler: in each step

- Basic Block:
  - straight-line code,
  - can only enter at beginning and left at end

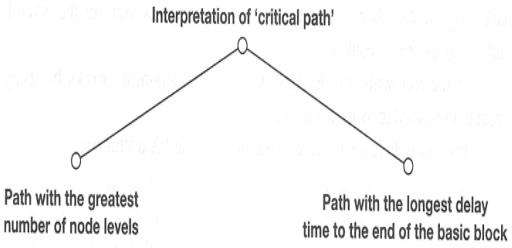


-Eligible Instructions are

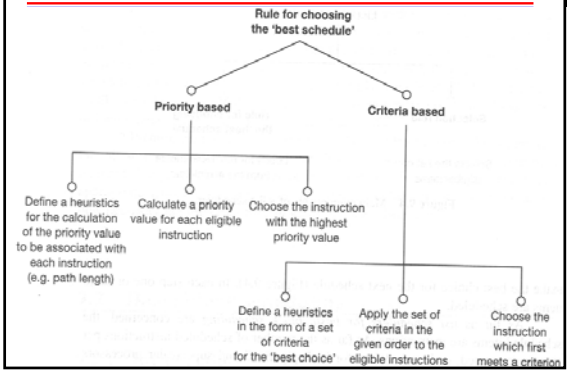
- dependency-free
  - **no predecessors in the Data Dependency Graph**
  - **or data to be produced by a predecessor node is already available**
    - **timing**: checking when the data will be ready
- hardware required resources are available

-Looks for the 'Best Choice' from the eligible set of instructions {critical path in Data Dependency Graph}

- Choose First: Schedule instructions in the critical path
- Choose Next: others



Heuristic Rule for choosing the next instructions



Set of criteria

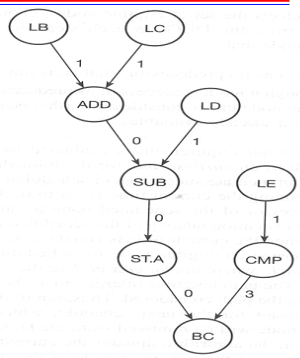
- 1. whether there is an immediate successor that is dependent on the instruction in question
- 2. how many successors the instruction has
- 3. how long is the longest path from the instruction in question to the leaves

Case Study: compiler for IBM Power1

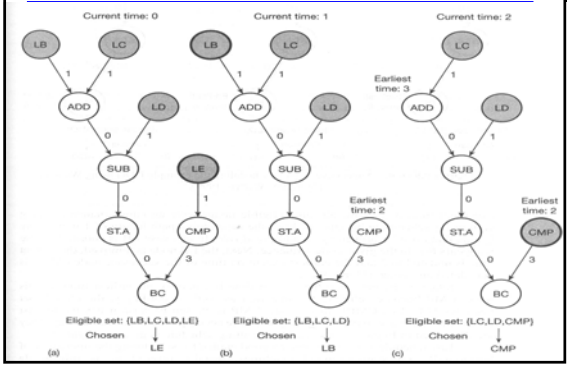
- Basic block approach: list scheduler
- Using Data Dependency Graph
- Critical path
  - → **the longest (in term of execution time)**
- Earliest time
  - → **when check the data for the instruction will be available**
- Latency value (on each arc of DDG)
  - → **how many time units the successors node have to wait before the result of the predecessor node becomes available**

Example program:
The intermediate code and the corresponding DDG

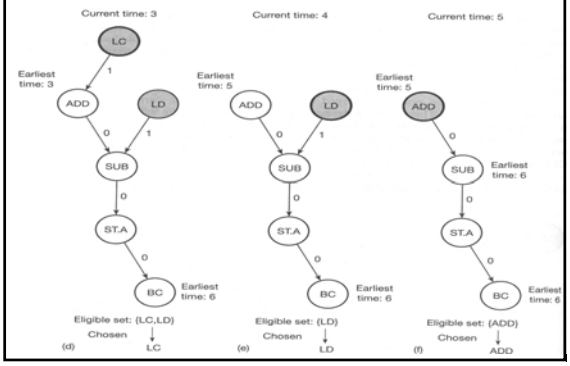
```
l     r100, b(r200)
l     r101, c(r200)
add   r102, r100, r101
l     r103, d(r200)
sub   r104, r102, r103
st    r104, a(r200)
l     r105, e(r200)
cmp   r106, r105, 0
bc    r106, . . .
```



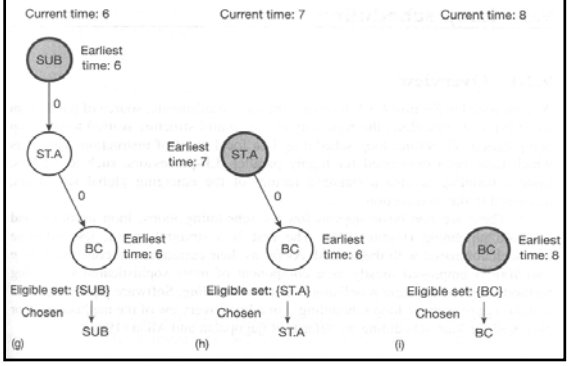
The first three steps in scheduling the example DDG using Warren's algoruthm



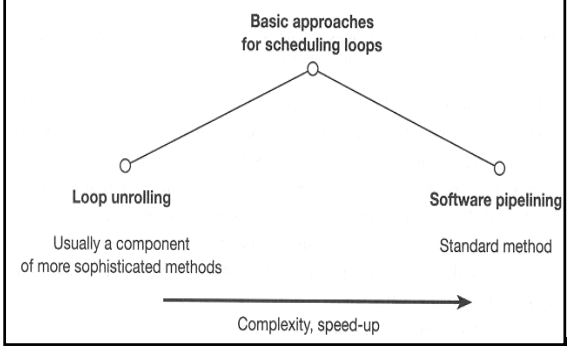
Successive steps in scheduling the example DDG



Successive steps in scheduling the example DDG



9.3 Loop Scheduling



-Loop Unrolling

- **for I = 1 to 3 do {**
- **b(I) = 2 * a(I)**
- **}**

- Loop Unrolled: basic concept
  - **b(1) = 2 * a(1)**
  - **b(2) = 2 * a(2)**
  - **b(3) = 2 * a(3)**
- save execution time at the expense of code length
- omitting inter-iteration loop updating code
  - **performance improvement**
- enlarging the basic block size
  - **can lead to more effective schedule with considerable speed up**

Simply unrolling is not practicable

- **when a loop has to be executed a large number of times**

- solution:
  - **unroll the loop a given number of time (e.g. 3)**
  - **→ larger basic block**
- Inter-iteration dependency
  - **only feasible if the required data becomes available in due time**

Speed-up produced by loop unrolling for the first 14 lawrence livermore loops

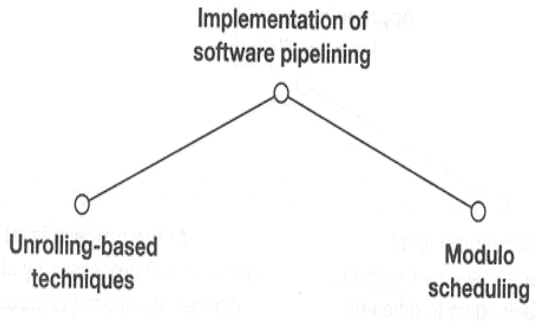
	Relative speed-up to the case when no unrolling is used		
Loop	*Unrolled by 2 blocks*	*Unrolled by 4 blocks*	*Unrolled by 8 blocks*
LLL *1	1.82	2.68	2.92
LLL *2	1.48	1.77	1.91
LLL *3	1.62	2.21	2.66
LLL †4	1.14	1.20	1.22
LLL #5	1.20	1.30	1.36
LLL #6	1.23	1.30	1.35
LLL *7	1.42	1.67	1.45
LLL †8	0.92	0.94	0.97
LLL *9	1.26	1.35	1.22
LLL *10	1.49	1.59	1.36
LLL #11	1.74	2.45	3.03
LLL *12	1.74	2.63	2.73
LLL †13	1.03	0.93	0.95
LLL †14	1.03	0.95	0.98
Aggregate	**1.34**	**1.50**	**1.56**
H-mean	**1.37**	**1.56**	**1.62**

* : no recurrences
# : recurrences, where memory hazards could be resolved during scheduling
† : recurrences, where memory hazard could not be resolved during scheduling

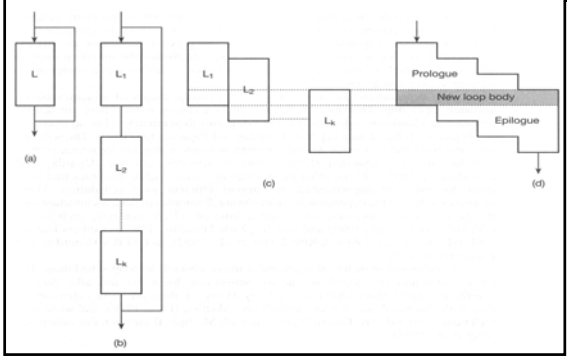
-Software Pipelining

- Software pipelining is an analogous term to hardware pipelining
- Software pipelining
  - **each pipeline stage uses one EU**
  - **each pipeline cycle execute one instruction**
- E.G.
  - **for I = 1 to 7 do {**
  - **b(I) = 2 * a(I)**
  - **}**

Basic methods for software pipelining



-Principle of the unrolling-based URPR sofeware pipelining method



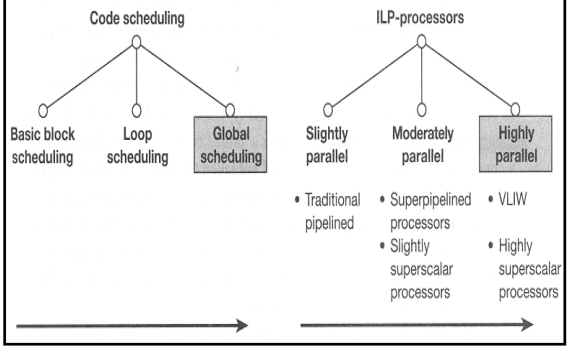
E.G. loop 7 times, 4 EU's VLIW, fmul takes 3 cycles

Cycle	*Iteration number*						
	1	2	3	4	5	6	7
c	load						
c+1	fmul	load					
c+2	decr	fmul	load				
c+3	nop	decr	fmul	load			
c+4	store	nop	decr	fmul	load		
c+5		store	nop	decr	fmul	load	
c+6			store	nop	decr	fmul	load
c+7				store	nop	decr	fmul
c+8					store	nop	decr
c+9						store	nop
c+10							store

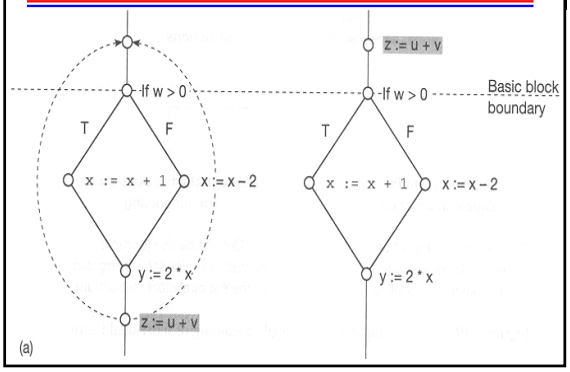
-Modulo scheduling //

- → find the repetitive character of the schedule
- guess the minimal required length of the new {partially unrolled} loop
  - **period of repetition**
- try schedule for this interval {period}
  - **taking into account data and resource dependencies**
- if the schedule is not feasible
  - **then the length {period} is increased**
- try again

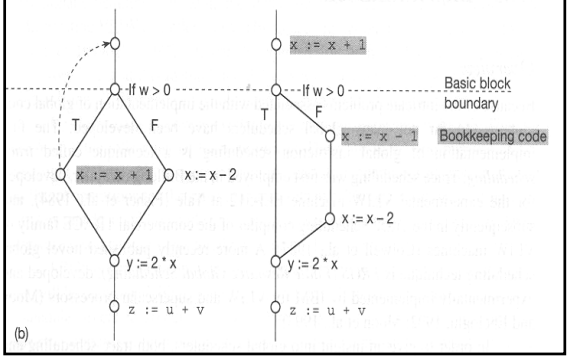
9.4 Global Scheduling:
Scheduling of individual instructions beyond basic blocks



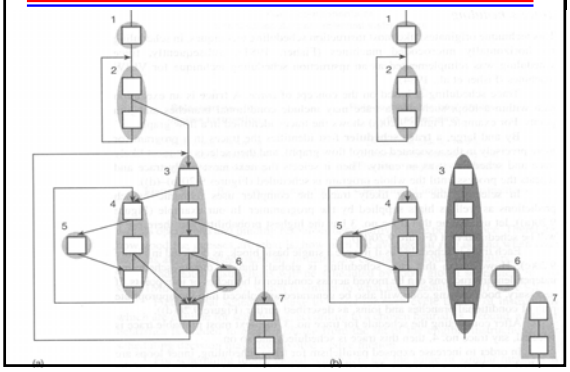
Moving up an independent instruction beyond a basic block boundary 'along both paths ' of a conditional branch



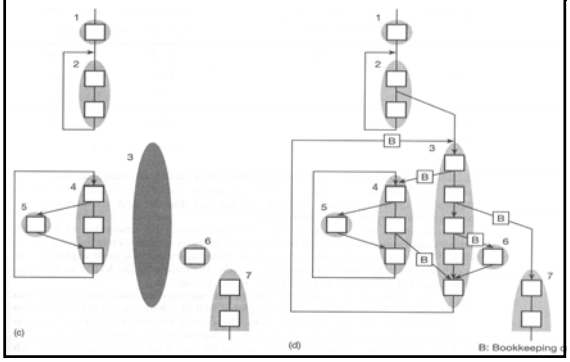
Moving up an independent instruction beyond a basic block boundary 'along a single path' of a conditional branch {guess T}



Trace Scheduling: flow graph, guess 3



Bookkeeping Code [B]



Finite Resource Global Scheduling: solving long compilation times and code explosion

- For VLIW
  - **3.7 Speed-up over Power1**
  - **49% Compilation time over PL.8**
  - **2.1 Code explosion over RISC**