Instruction Scheduling

Compilers for High Performance Architectures

Outline

- Resource modeling
- List scheduling
- Code regions
  - Extended basic block
  - Traces
  - Superblocks
  - Hyperblock

Machine Information

- Each step of code generation requires knowledge of the machine
- What does the code generator need to know about the target processor?
  - Structural information? -> No
  - For each opcode
    - What registers can be accessed as each of its operands
    - Other operand encoding limitations
  - Operation latencies
    - Read inputs, write outputs
  - Resources utilized
    - Which ones, when
- **Machine description**
  Each unit specific opcode has 3 properties
  - IO format
  - Latency
  - Resource usage

IO Format

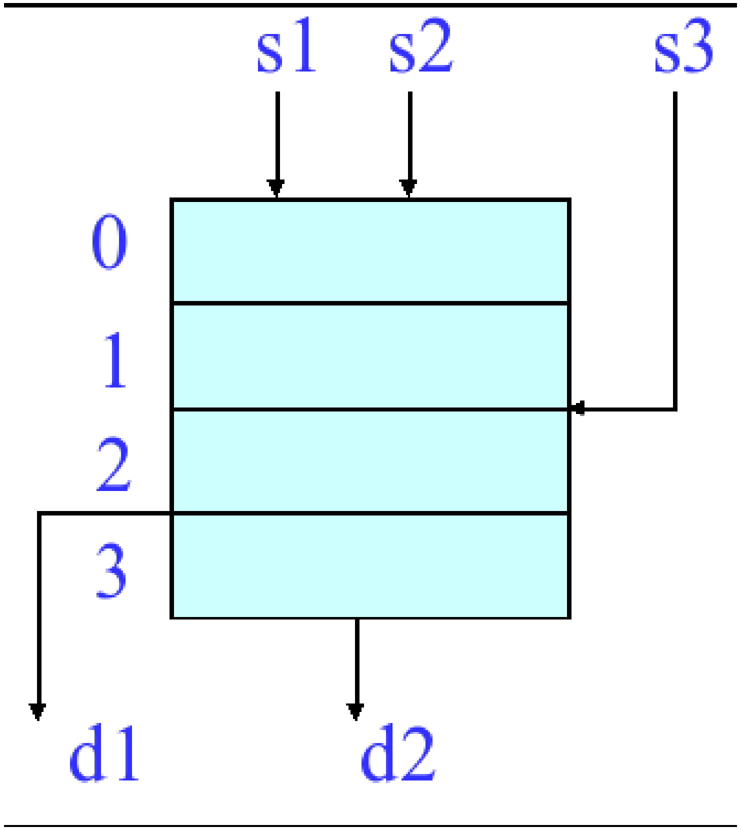
- Registers, register files
  - Number, width, static or rotating
  - Read-only (hardwired 0) or read-write
- Operation
  - Number of source and destinations
  - Predicated or not
  - For each source / destination
    - What register file(s) can be read/written
    - Can be a literal? if so, how big?

Latency Information

- Multiply takes 3 cycles … no, it’s not that simple!!!
- Differential input/output latencies
  - Earliest read latency for each source operand
  - Latest read latency for each source operand
  - Earliest write latency for each destination operand
  - Latest write latency for each destination operand
- Why all this?
  - Unexpected events may make operands arrive late or be produced early
  - Compound op: part may finish early or start late
  - Instruction re-execution by
    - Exception handlers
    - Interupt handlers
- Many processors follow a simpler model where
  - Earliest read latency = Latest read latency
  - Earliest write latency = Latest write latency

Latency Information

- Multiply and Add
  - mpadd d1, d2, s1, s2, s3
    - d1 = s1 * s2
    - d2 = s3 + d1
- Early/late read (sources) or write (destintation)
  - s1: 0, 2
  - s2: 0, 2
  - s3: 2, 2
  - d1: 2, 3
  - d2: 2, 4

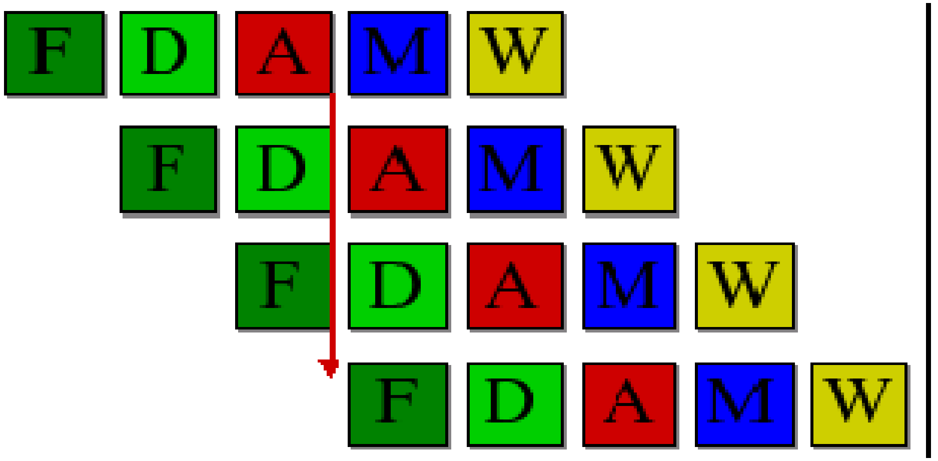


Memory Serialization Latency

- Ensuring the proper ordering of dependent memory operations
  - Not the memory latency
  - But, point in the memory pipeline where 2 ops are guaranteed to be processed in sequential order
- Page fault –> memory op is re-executed, so need
  - Earliest mem serialization latency
  - Latest mem serialization latency
- Remember
  - Compiler will use this, so any 2 memory ops that cannot be proven independent, must be separated by mem serialization latency.

Branch Latency

- Time relative to the initiation time of a branch at which the target of the branch is initiated
- What about branch prediction?
  - Can reduce branch latency
  - But, may not make it 1
- Make delay slots visible to the compiler



Resources

- Any aspect of the target processor for which over-subscription is possible
    - Scheduler must pick conflict free combinations
- 3 kinds of machine resources
    - Hardware resources are hardware entities that would be occupied or used during the execution of an opcode
        - Integer ALUS, pipeline stages, register ports, busses, etc.
    - Abstract resources are conceptual entities that are used to model operation conflicts or sharing constraints that do not directly correspond to any hardware resource
        - Sharing an instruction field
    - Counted resources are identical resources such that k are required to do something
        - Any 2 input busses

Reservation Tables

For each opcode, the resources used at each cycle relative to its initiation time are specified in the form of a table

Res1, Res2 are abstract resources to model issue constraints

relative time	Res1	Res2	ALU	MPY	Resultbus
0	X		X		
1					X

Integer add

relative time	Res1	Res2	ALU	MPY	Resultbus
0		X		X	
1				X	
2					X

Non-pipelined multiply

relative time	Res1	Res2	ALU	MPY	Resultbus
0	X	X	X		
1					X

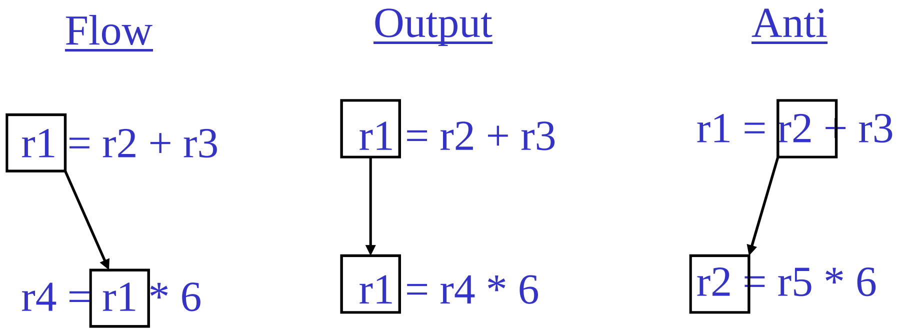
Load, uses ALU for addr calculation, can’t issue load with add or multiply

Code Scheduling

- Scheduling constraints
  - Processor resources –> modeled by machine description
  - Dependences between operations
    - Data, memory, control
- Processor resources
  - Manage using resource usage map (RU_map)
    - When each resource will be used by already scheduled ops
  - Considering an operation at time t
    - See if each resource in reservation table is free
  - Schedule an operation at time t
    - Update RU_map by marking resources used by op busy

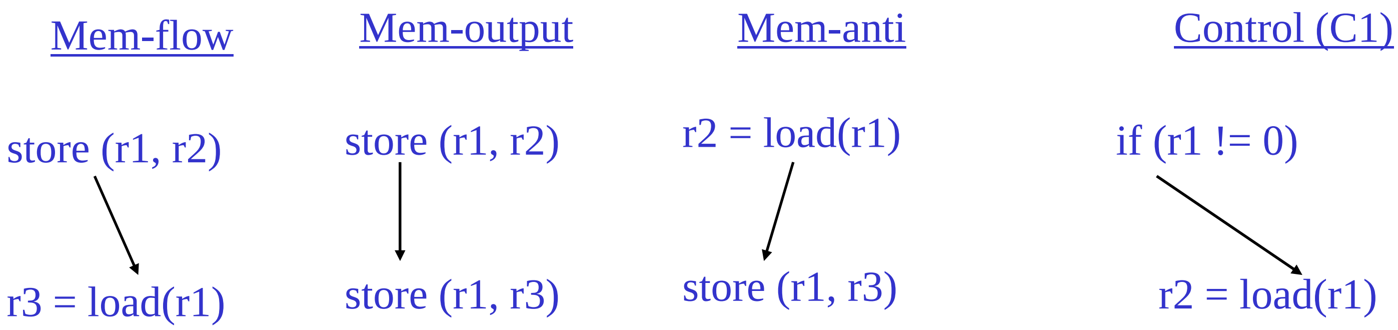
Data Dependences

- Data dependences
  - If 2 operations access the same register, they are dependent
  - However, only keep dependences to most recent producer/consumer as other edges are redundant
  - Types of data dependences
    - Flow (RAW)
    - Output (WAW)
    - Anti (WAR)



More Dependences

- Memory dependences
  - Similar as register, but through memory
  - Memory dependences may be certain or maybe
- Control dependences
  - Branch determines whether an operation is executed or not
  - Operation must execute after/before a branch



Mem-flow

store (r1, r2)

r3 = load(r1)

Mem-output

store (r1, r2)

store (r1, r3)

Mem-anti

r2 = load(r1)

store (r1, r3)

Control (C1)

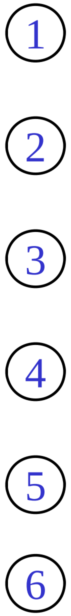
if (r1 != 0)

r2 = load(r1)

Dependence Graph

- Represent dependences between operations in a block via a DAG
  - Nodes = operations
  - Edges = dependences
- Single-pass traversal required to insert dependences
- Example

```
    1: r1 = load(r2)
    2: r2 = r1 + r4
    3: store (r4, r2)
    4: p1 = cmpp (r2 < 0)
    5: branch if p1 to BB3
    6: store (r1, r2)
BB3:
```



Dependence Edge Latencies

- Edge latency = minimum number of cycles necessary between initiation of the predecessor and successor in order to satisfy the dependence
- Register flow dependence, a → b
  - Latest_write(a) – Earliest_read(b)
- Register anti dependence, a → b
  - Latest_read(a) – Earliest_write(b) + 1
- Register output dependence, a → b
  - Latest_write(a) – Earliest_write(b) + 1
- Negative latency
  - Possible, means successor can start before predecessor
  - We will only deal with latency >= 0 -> MAX any latency to 0

Dependence Edge Latencies (2)

- Memory dependences, a → b (all types, flow, anti, output)
  - latency = latest_serialization_latency(a) – earliest_serialization_latency(b) + 1
  - Prioritized memory operations
    - Hardware orders memory ops by order in MultiOp
    - Latency can be 0 with this support
- Control dependences
  - branch → b
    - Op b cannot issue until prior branch completed
    - latency = branch_latency
  - a → branch
    - Op a must be issued before the branch completes
    - latency = 1 – branch_latency (can be negative)
      - Can fill delay slot
    - conservative, latency = MAX(0, 1-branch_latency)
      - Cannot fill delay slot

Class Problem

machine model

min/max read/write latencies

add: src 0/1
dst 1/1

mpy: src 0/2
dst 2/3

load: src 0/0
dst 2/2
sync 1/1

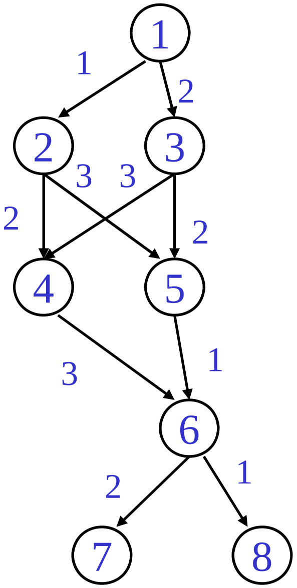
store: src 0/0
dst -
sync 1/1

1. Draw dependence graph
2. Label edges with type and latencies

```
r1 = load(r2)
r2 = r2 + 1
store (r8, r2)
r3 = load(r2)
r4 = r1 * r3
r5 = r5 + r4
r2 = r6 + 4
store (r2, r5)
```

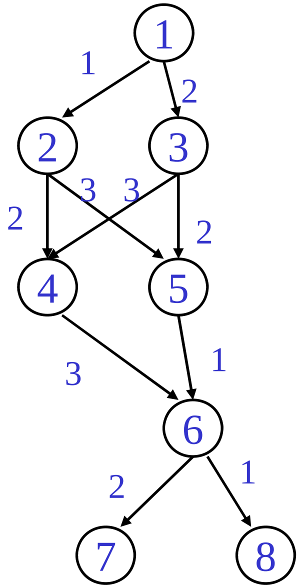
Dependence Graph Properties - Estart

- Estart = earliest start time, (as soon as possible - ASAP)
  - Schedule length with infinite resources (dependence height)
  - Estart = 0 if node has no predecessors
  - Estart = MAX(Estart(pred) + latency) for each predecessor node
  - Example



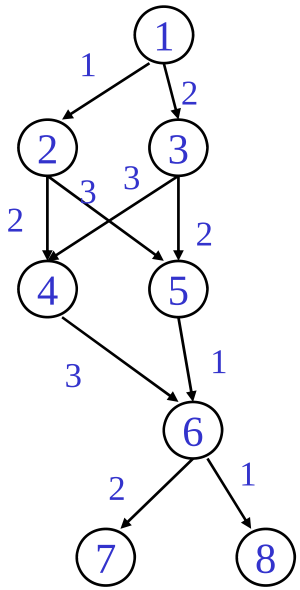
Lstart

- Lstart = latest start time, ALAP
  - Latest time a node can be scheduled without increasing schedule length
  - Lstart = MaxEstart if node has no successors
  - Lstart = MIN(Lstart(succ) - latency) for each successor node
  - Example



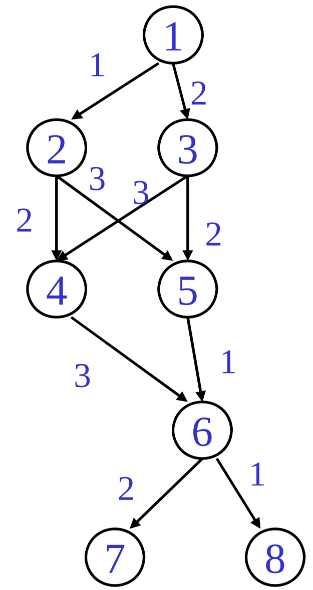
Slack

- Slack = measure of the scheduling freedom
  - Slack = Lstart – Estart for each node
  - Larger slack means more mobility
  - Example

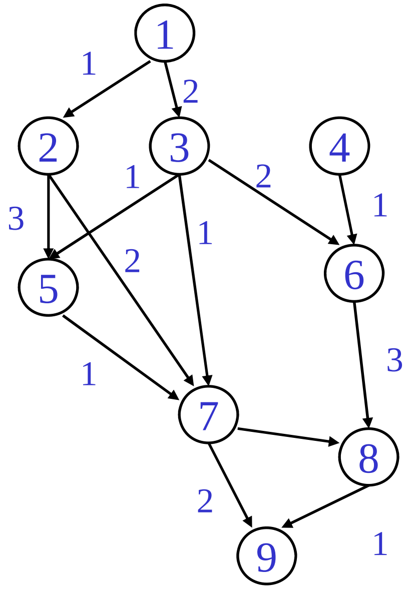


Critical Path

- Critical operations = Operations with slack = 0
  - No mobility, cannot be delayed without extending the schedule length of the block
  - Critical path = sequence of critical operations from node with no predecessors to exit node, can be multiple critical paths



Class Problem



Node	Estart	Lstart	Slack
1			
2			
3			
4			
5			
6			
7			
8			
9			

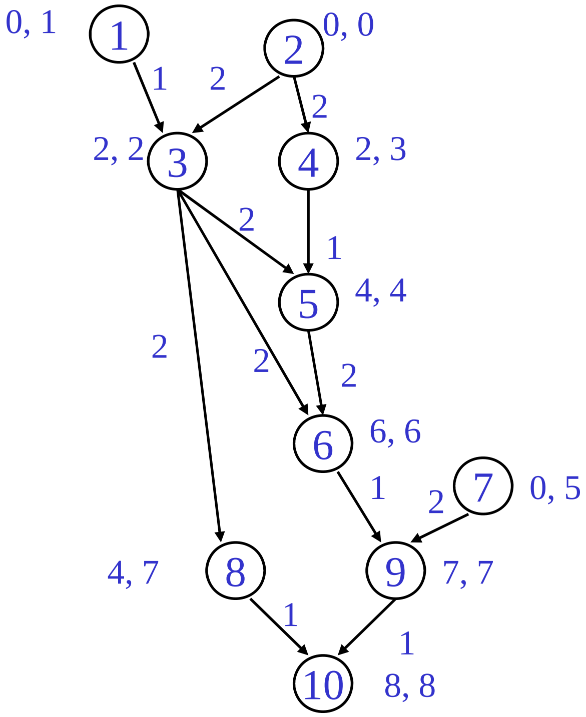
Critical path(s) =

Operation Priority

- Priority – Need a mechanism to decide which ops to schedule first (when you have multiple choices)
- Common priority functions
  - Height –> Distance from exit node
    - Give priority to amount of work left to do
  - Slackness –> inversely proportional to slack
    - Give priority to ops on the critical path
  - Register use –> priority to nodes with more source operands and fewer destination operands
    - Reduces number of live registers
  - Uncover –> high priority to nodes with many children
    - Frees up more nodes
  - Original order –> when all else fails

Height-Based Priority

- Height-based is the most common
  - priority(op) = MaxLstart – Lstart(op) + 1



op	priority
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

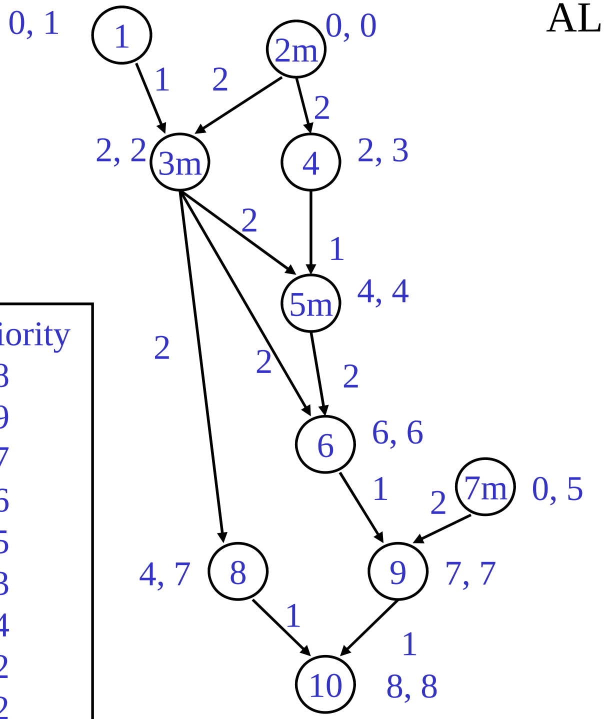
List Scheduling (Cycle Scheduler)

- Build dependence graph, calculate priority
- Add all ops to UNSCHEDULED set
- Schedule operations at earliest available cycle
  - In priority order, satisfying dependencies

```
time = -1
while (UNSCHEDULED is not empty)
  time++
  READY = UNSCHEDULED ops with satisfied incoming dependences
  Sort READY using priority function
  For each op in READY (highest to lowest priority)
    op can be scheduled at current time?
    Yes, schedule it, op.issue_time = time
      Mark resources busy in RU_map relative to issue time
      Remove op from UNSCHEDULED/READY sets
    No, continue
  (proceed to next cycle)
```

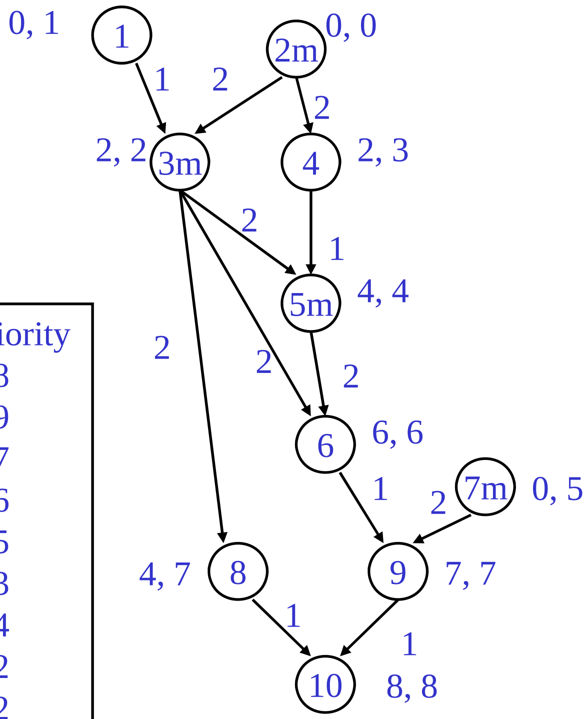
Cycle Scheduling Example

Machine: 2 issue, 1 memory port, 1 ALU
Memory port = 2 cycles, non-pipelined
ALU = 1 cycle



op	priority
1	8
2	9
3	7
4	6
5	5
6	3
7	4
8	2
9	2
10	1

Cycle Scheduling Example (2)



op	priority
1	8
2	9
3	7
4	6
5	5
6	3
7	4
8	2
9	2
10	1

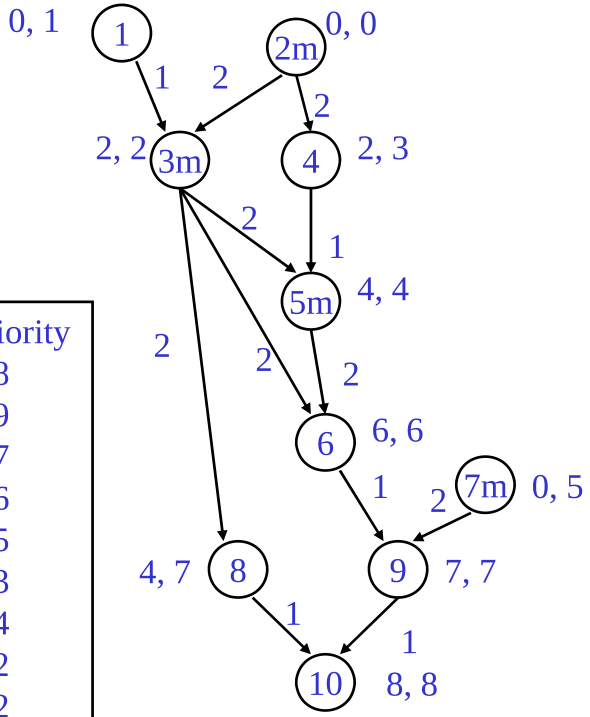
RU_map

time	ALU	MEM
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Schedule

time	Ready	Placed
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Cycle Scheduling Example (3)



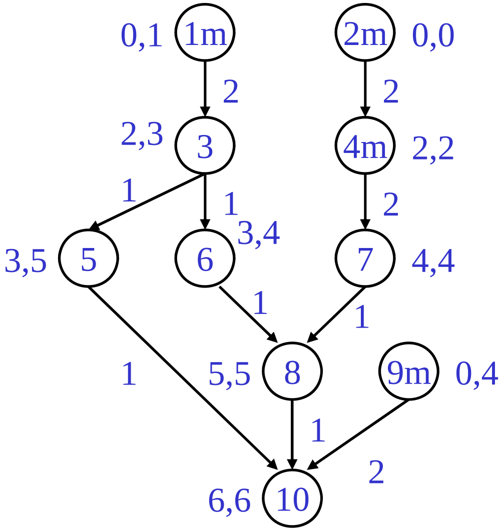
op	priority
1	8
2	9
3	7
4	6
5	5
6	3
7	4
8	2
9	2
10	1

Schedule

time	Ready	Placed
0	1,2,7	1,2
1	7	-
2	3,4,7	3,4
3	7	-
4	5,7,8	5,8
5	7	-
6	6,7	6,7
7	-	
8	9	9
9	10	10

Class Problem

Machine: 2 issue, 1 memory port, 1 ALU
Memory port = 2 cycles, pipelined
ALU = 1 cycle



1. Calculate height-based priorities
2. Schedule using cycle scheduler

Code Regions

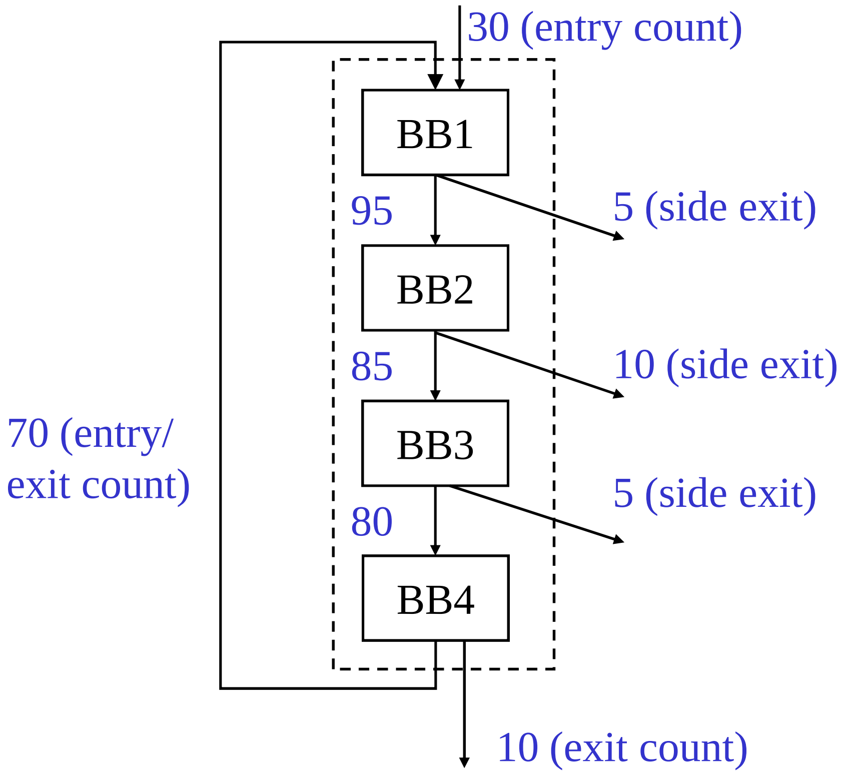
- Region: A collection of operations that are treated as a single unit by the compiler
  - Examples
    - Basic block
    - Procedure
    - Body of a loop
  - Properties
    - Connected subgraph of operations
    - Control flow is the key parameter that defines regions
    - Hierarchically organized
- Problem
  - Basic blocks are too small (3-5 operations)
    - Hard to extract sufficient parallelism
  - Procedure control flow too complex for many compiler transformations
    - Plus only parts of a procedure are important (90/10 rule)

Code regions (II)

- Want
  - Intermediate sized regions with simple control flow
  - Bigger basic blocks would be ideal !!
  - Separate important code from less important
  - Optimize frequently executed code at the expense of the rest
- Solution
  - Define new region types that consist of multiple BBs
  - Profile information used in the identification
  - Sequential control flow (sort of ...)
  - Pretend the whole region is a single basic block
  - May break control dependencies -> speculative scheduling

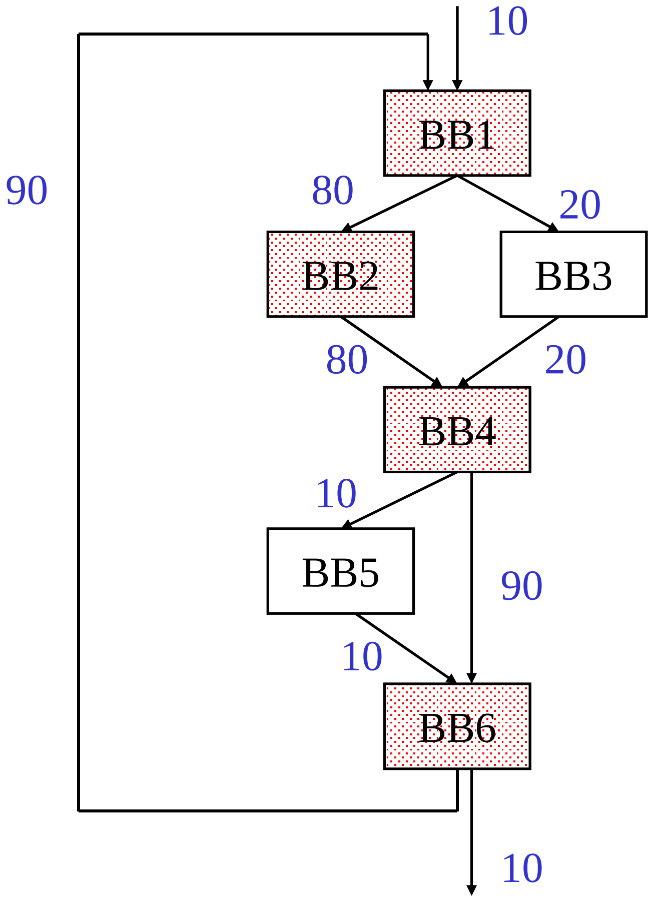
Extended basic block

- Definition: Maximal sequence of instructions beginning with a leader, containing no other join nodes (other than the first node).
  - Has a single entry
  - Has multiple exists
- Can be seen as a tree, with the entry point as the root
- Appears naturally when unrolling non-counted loops

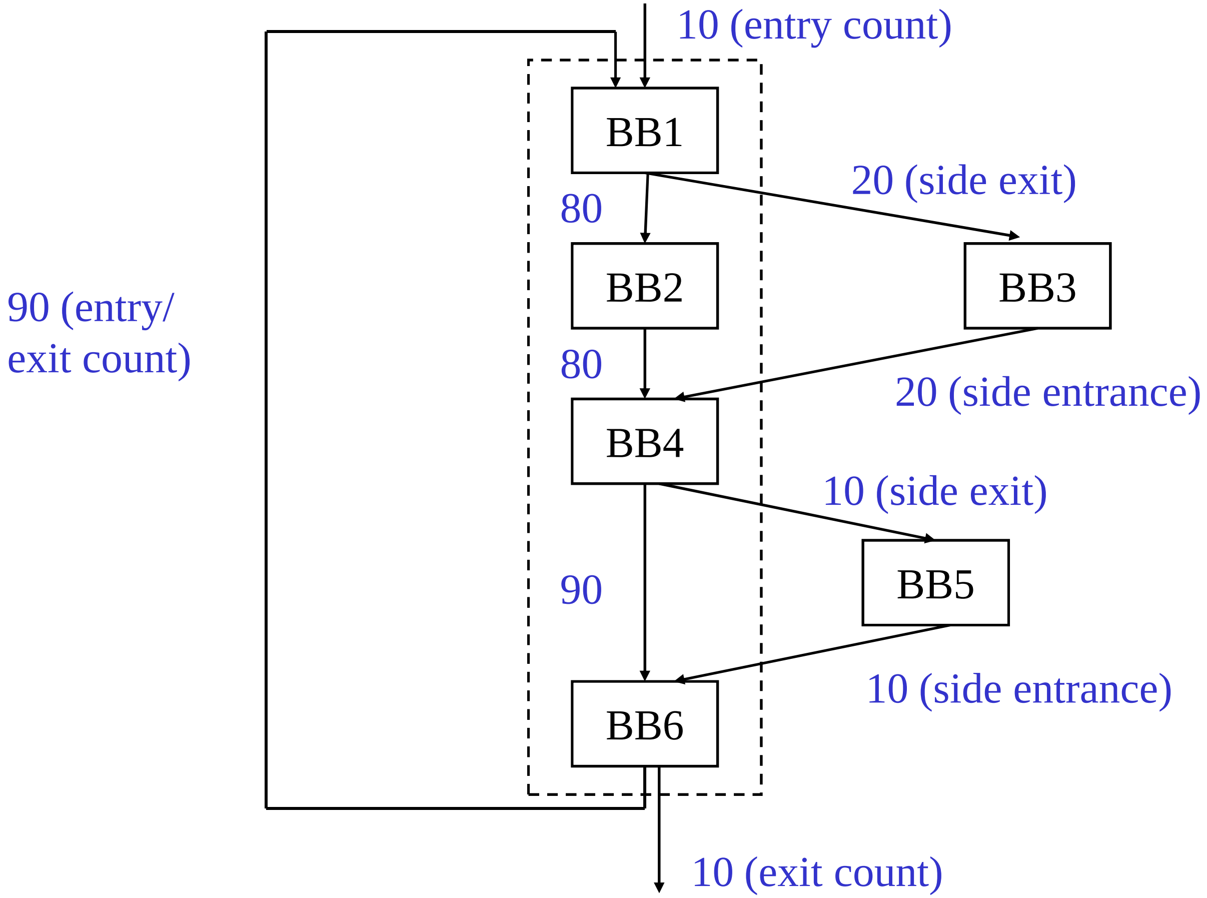


Trace

- Trace - Linear collection of basic blocks that tend to execute in sequence
  - "Likely control flow path"
  - Acyclic (outer backedge ok)
- Side entrance – branch into the middle of a trace
- Side exit – branch out of the middle of a trace
- Compilation strategy
  - Compile assuming path occurs 100% of the time
  - Patch up side entrances and exits afterwards
    - Add fix-up code to correct miss-speculated instructions
- Motivated by scheduling (i.e., trace scheduling)

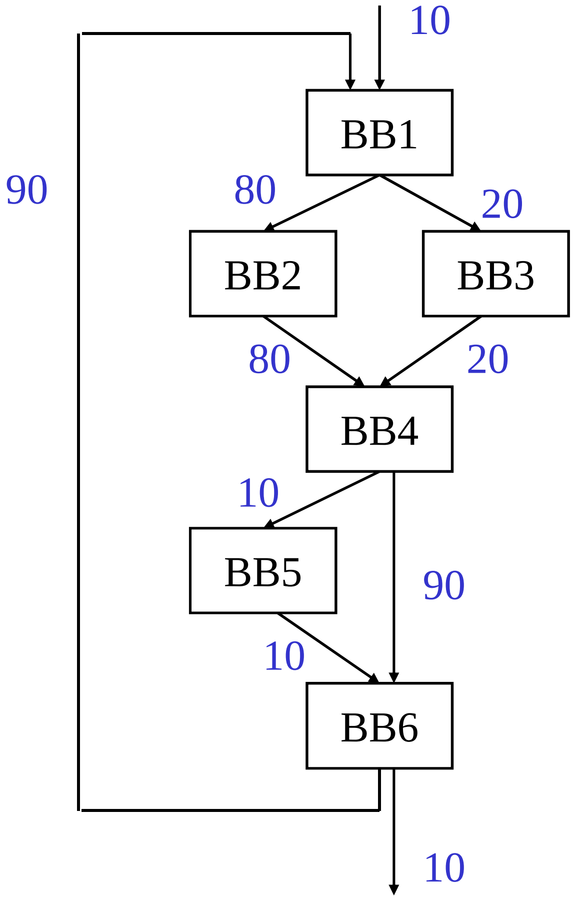


Linearizing a Trace



Issues With Selecting Traces

- Acyclic
  - Cannot go past a backedge
- Trace length
  - Longer = better ?
    - Not always !
- On-trace / off-trace transitions
  - Maximize on-trace
  - Minimize off-trace
- Compile assuming on-trace is 100% (ie single BB)
  - Penalty for off-trace -> fix-up code
- Tradeoff (heuristic)
  - Length
  - Likelihood remain within the trace



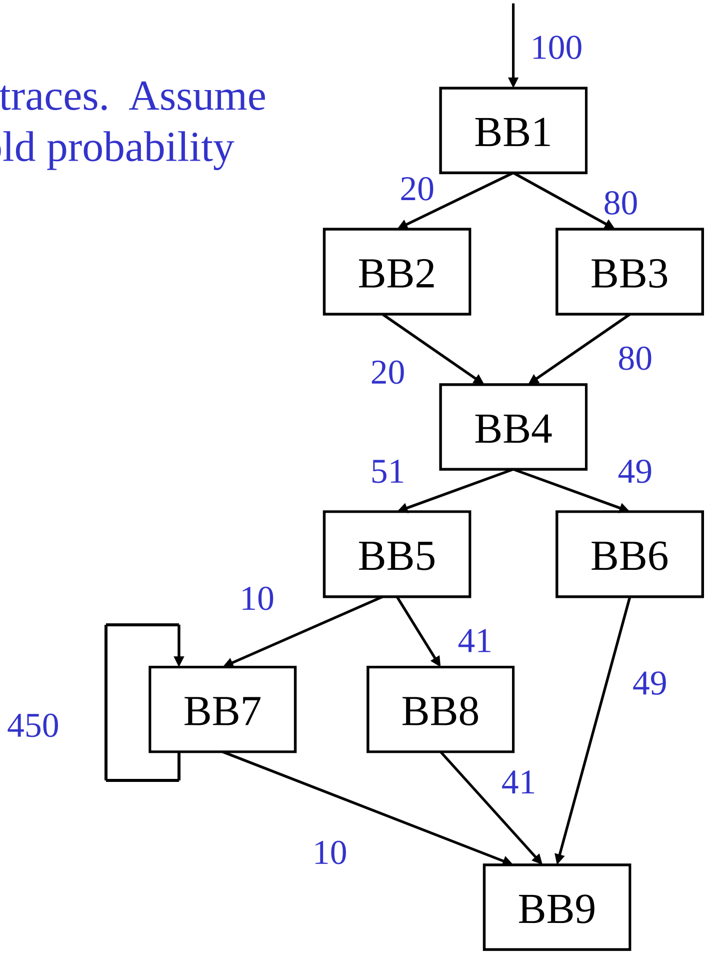
Trace construction

- Start trace from a seed
  - Select heaviest BB in CFG
  - Select heaviest Edge in CFG
    - More accurate
- Extend trace foward
  - Choose best successor
  - Set a threshold transition probability
- Extend trace backwards
  - Choose best predecessor
  - Set a threshold transition probability

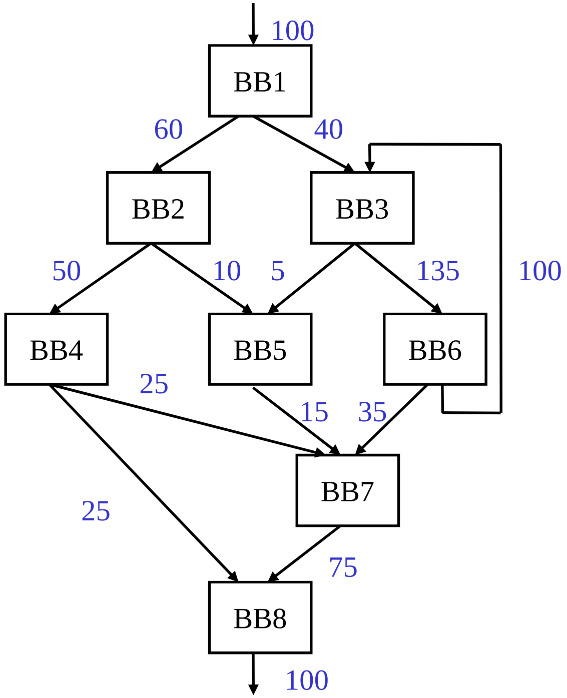
- Notes on this algorithm
  - BB only allowed in 1 trace
  - Min weight for seed to be chosen (ie executed 100 times)
  - THRESHOLD
    - controls off-trace probability
    - 60-70% found best

Class Problem 1

Find the traces. Assume a threshold probability of 60%.



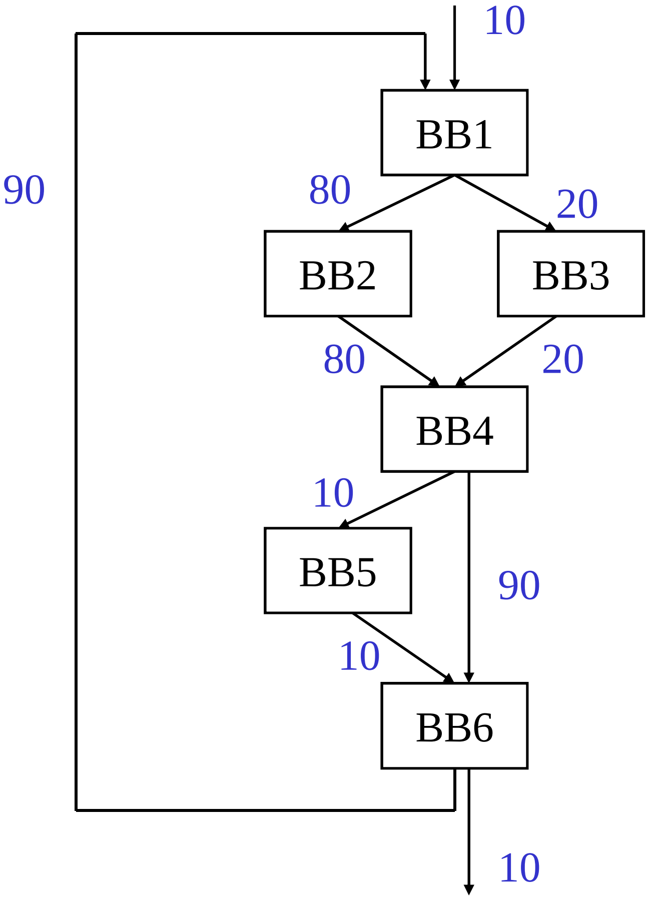
Class Problem 2



Find the traces. Assume
a threshold probability of 60%.

The need to go beyond traces

- Treat trace as a big BB
  - Transform trace ignoring side entrance/exits
  - Insert fixup code
    - aka bookkeeping
    - aka compensation code
- Side entrance fixup is more painful
  - Sometimes not possible so transform not allowed
- Solution
  - Eliminate side entrances
    - The ***superblock*** is born

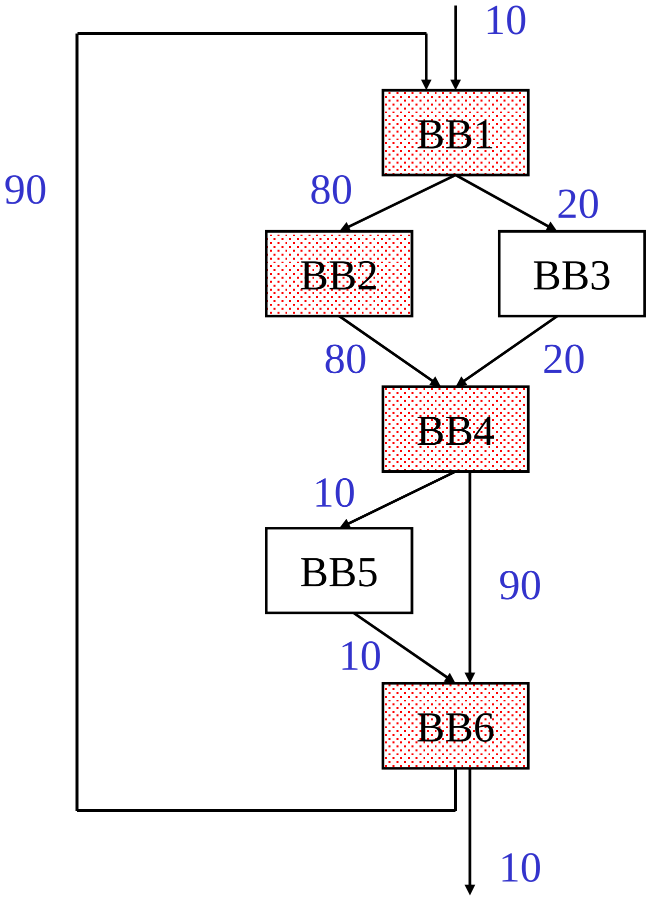


Superblocks

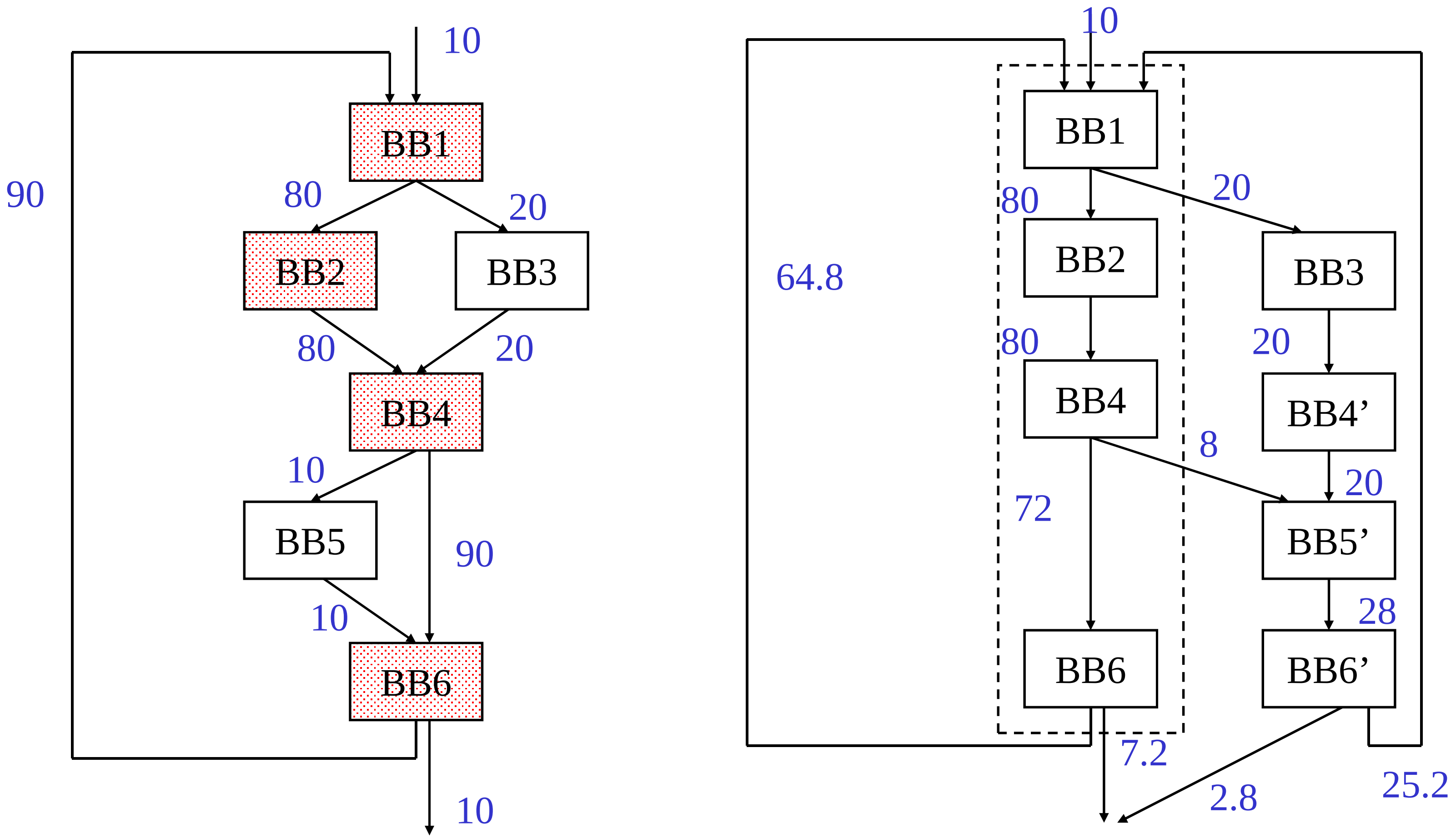
- Superblock - Linear collection of basic blocks that tend to execute in sequence *in which control flow may only enter at the first BB*
  - Represents the most likely control flow path
  - Acyclic (outer backedge ok)
- Is a trace with no side entrances
  - Side exits still exist
- Superblock formation
  - 1. Trace selection
  - 2. Eliminate side entrances

Tail replication

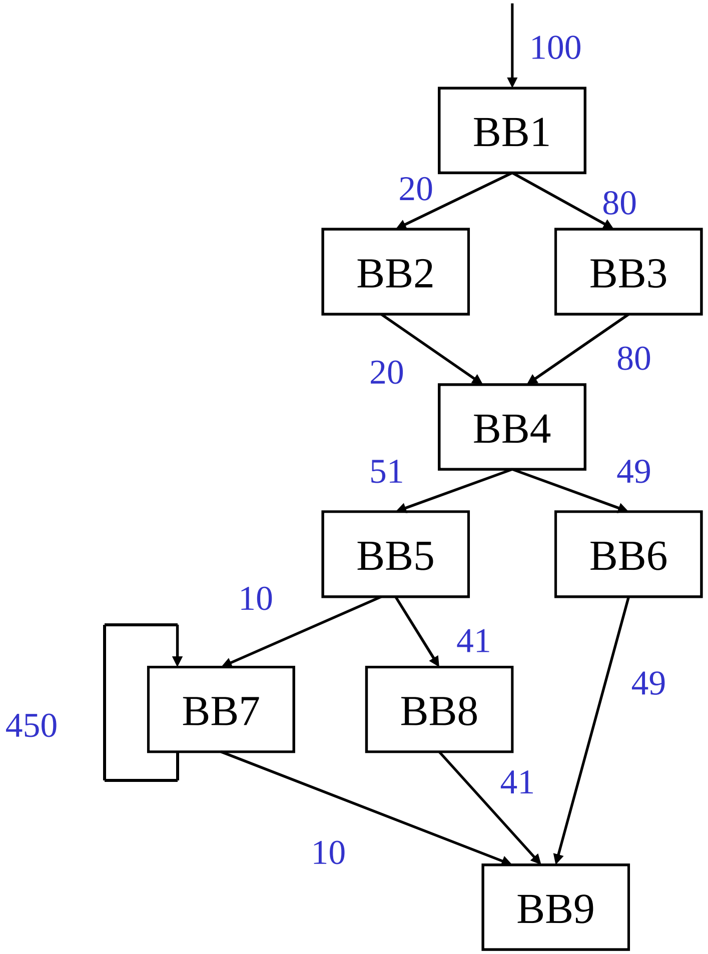
- To eliminate all side entrances replicate the “tail” portion of the trace
  - Identify first side entrance
  - Replicate all BB from the target to the bottom
  - Redirect all side entrances to the duplicated BBs
    - Copy each BB only once
- Max code expansion = 2x-1 where x is the number of BB in the trace
- Must adjust profile information afterwards



Superblock Formation



Class Problem 3



Create the superblocks, trace threshold is 60%

Issues with Superblocks

- Central tradeoff
  - Side entrance elimination
    - Compiler complexity
    - Compiler effectiveness
  - Code size increase
- Apply intelligently
  - Most frequently executed BBs are converted to SBs
    - Set upper limit on code expansion
    - 1.0 – 1.10x are typical code expansion ratios from SB formation
- Must still patch side exits
  - How to remove these?

The Hyperblock

- Hyperblock - Collection of basic blocks in which control flow may only enter at the first BB. All internal control flow is eliminated via if-conversion
  - Represents the most likely control flow path
  - Must be acyclic (outer backedge ok)
  - Multiple intersecting traces with no side entrances
  - Side exits still may exist
- Hyperblock formation
  - 1. Block selection
  - 2. Tail duplication
  - 3. If-conversion

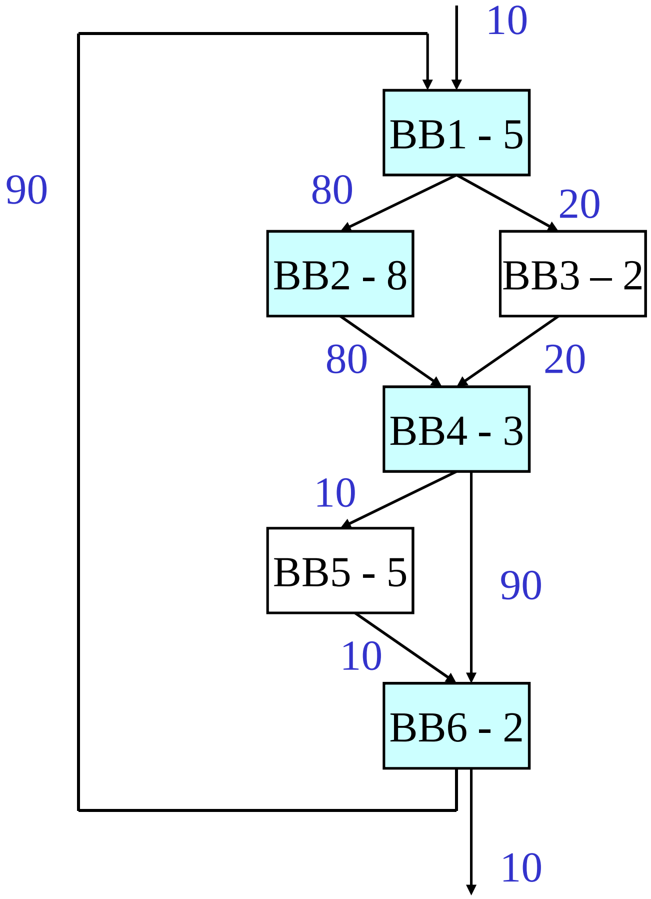
Block selection for hyperblocks

- Select subset of BBs for inclusion in HB
- Difficult problem
  - Weighted cost/benefit function
  - Costs
    - Height overhead
    - Resource overhead
    - Hazard overhead
  - Benefits
    - Branch elimination benefit
    - Compensation code elimination

Block selection heuristics

- Create a trace → set it as “main path”
- Select other blocks that are “compatible” with main path
  - Consider each BB by itself for simplicity
    - Compute priority for other BB’s
    - Normalize against main path
  - Block Selection Value for BB(i)
    - BSVi = (K x (BBi_weight / BBi_size)
      x (MainPath_size / MainPath_weight) x BBi_char)
    - weight = execution frequency
    - size = number of operations
    - char = characteristic value of each BB
      - Max value = 1
      - Hazardous instructions reduce this to 0.5, 0.25, …
    - K = constant to represent processor issue rate
  - Include BB when BSVi > Threshold

Example - Step 1 - Block Selection



main path = 1,2,4,6

num_ops = 5 + 8 + 3 + 2 = 18

weight = 80

Calculate the BSVs for BB3, BB5 assuming no hazards, K = 4

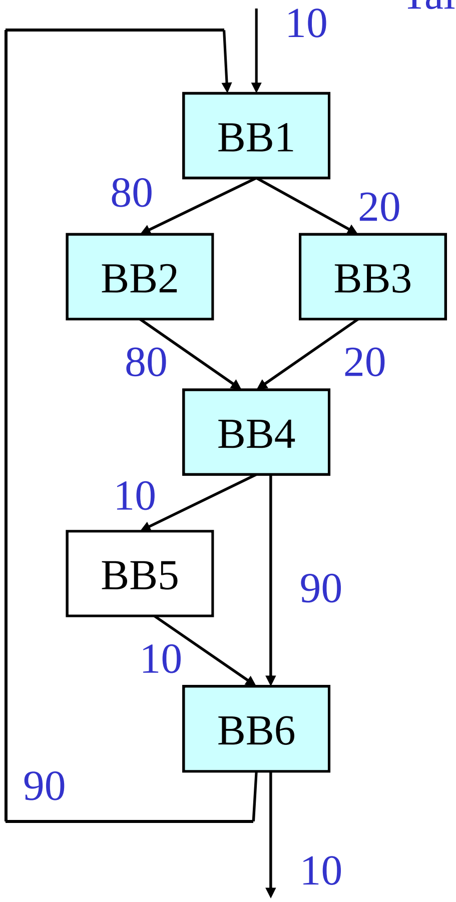
BSV3 = 4 x (20 / 2) x (18 / 80) = 9

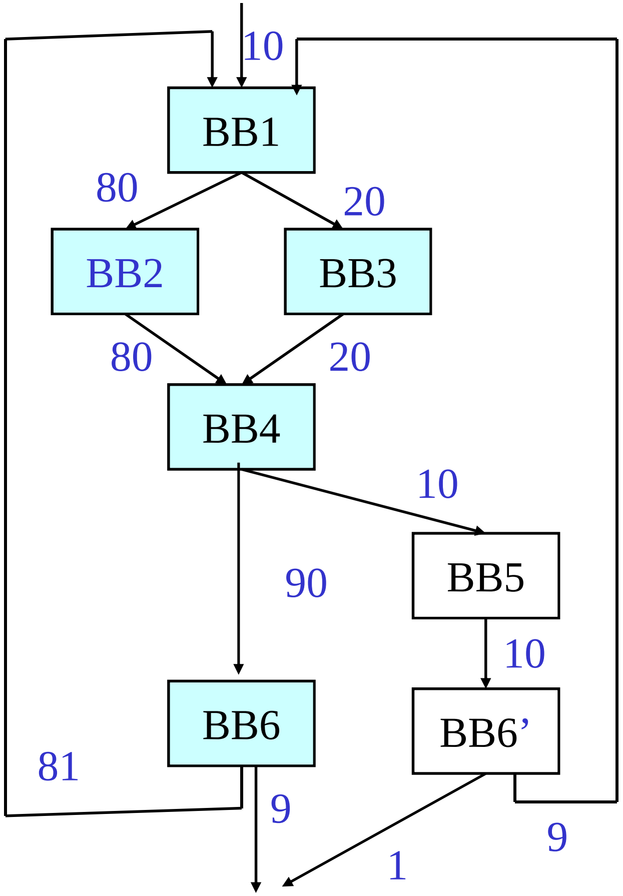
BSV5 = 4 x (10 / 5) x (18 / 80) = 1.8

If Threshold = 2.0, select BB3 along with main path

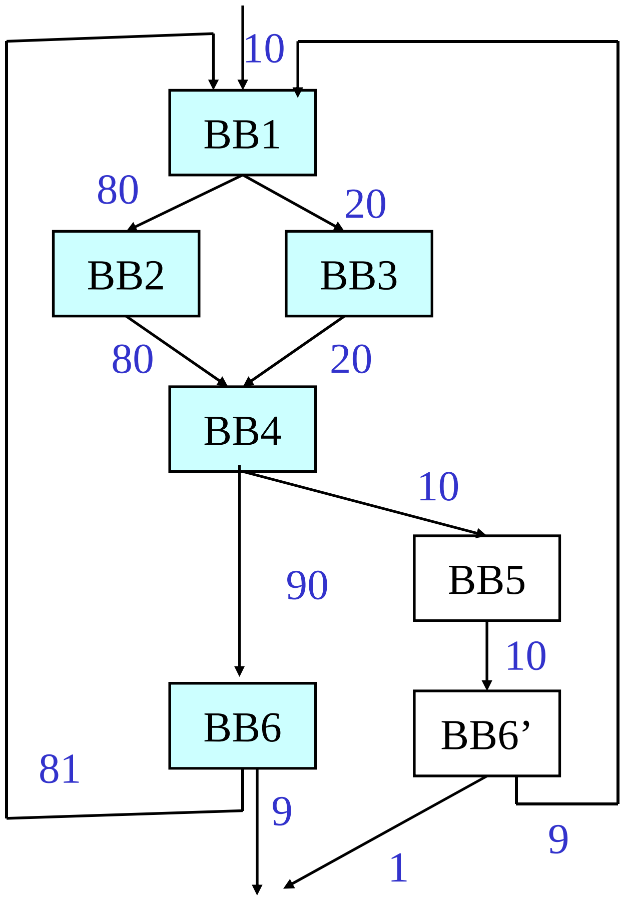
Example - Step 2 - Tail Duplication

Tail duplication same as with Superblock formation

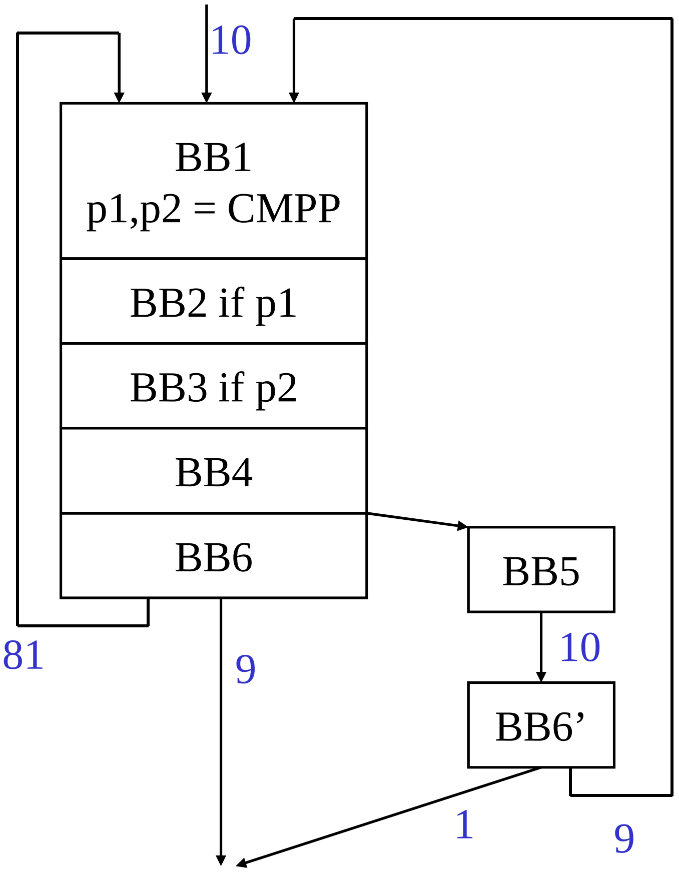




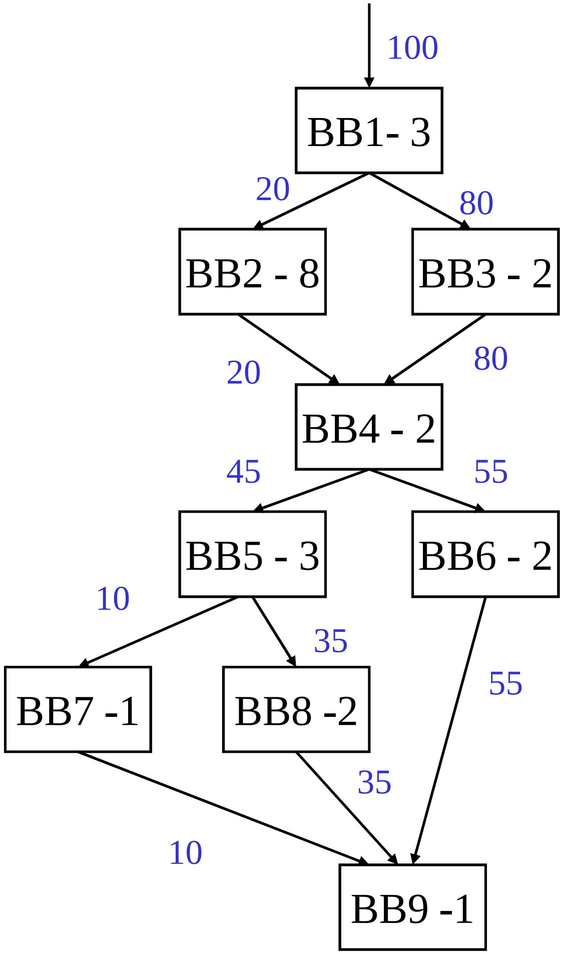
Example - Step 3 – If-conversion



If-convert intra-HB branches only!!



Class Problem



Form the HB for this subgraph
Assume K = 4, BSV Threshold = 2