EECS 583 – Class 10
ILP Optimization and Intro. to Code Generation

University of Michigan

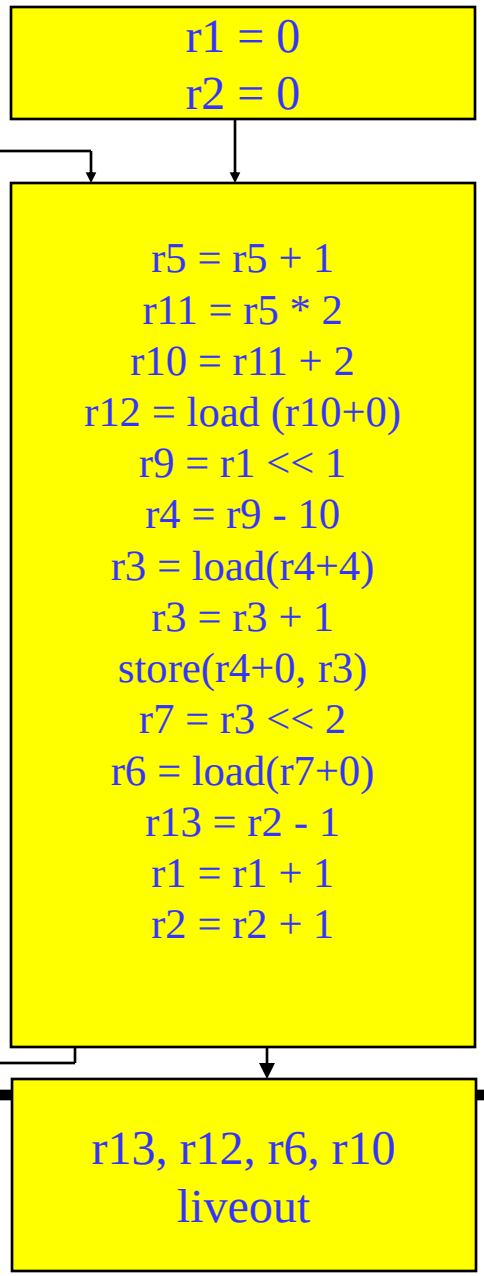
October 10, 2011

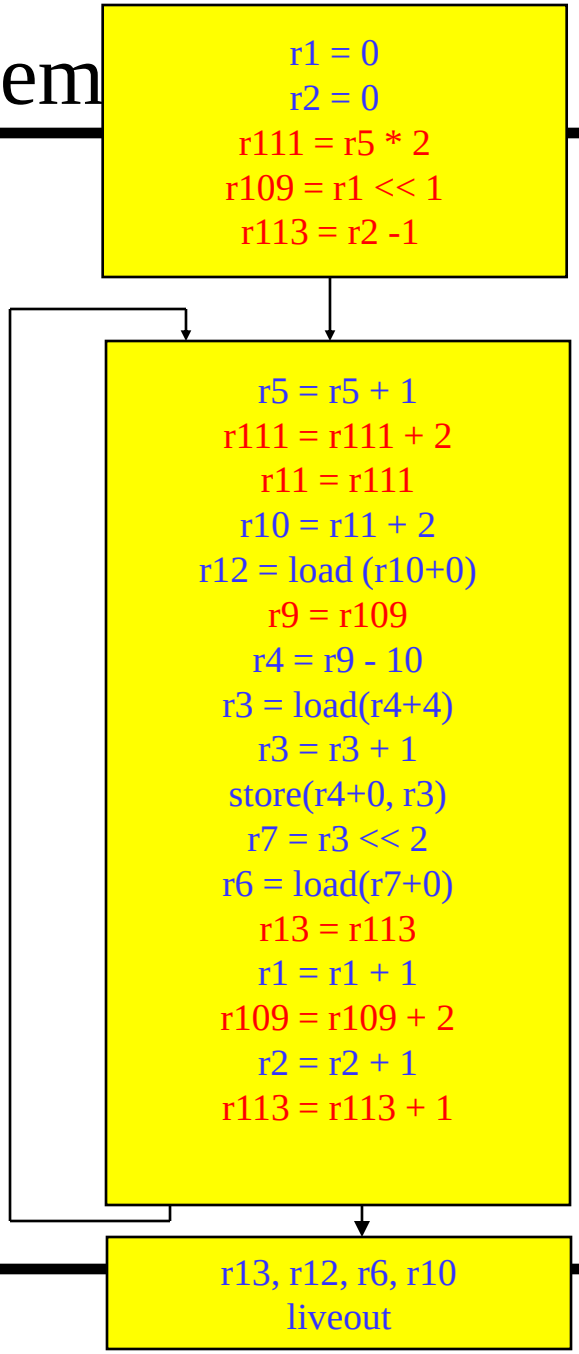
Reading Material + Announcements

- List of references for each project area available on course website
- Places to find papers on compilers
  - Conferences: PLDI, CGO, VEE, CC, Micro, Asplos
  - Journals: ACM Transactions on Computer Architecture and Code Optimization, Software Practice & Experience, IEEE Transactions on Computers, ACM Transactions on Programming Languages and Systems
- Today's class
  - "Machine Description Driven Compilers for EPIC Processors", B. Rau, V. Kathail, and S. Aditya, HP Technical Report, HPL-98-40, 1998. (long paper but informative)
- Next class
  - "The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors," P. Chang et al., IEEE Transactions on Computers, 1995, pp. 353-370.

From Last Time: Class Problem

Optimize this applying induction var str reduction





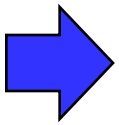
Note, after copy propagation, r10 and r4 can be strength reduced as well.

From Last Time: Class Problem

Assume: + = 1, * = 3

operand	0	0	0	1	2	0
arrival times	r1	r2	r3	r4	r5	r6

```
r10 = r1 * r2
r11 = r10 + r3
r12 = r11 + r4
r13 = r12 – r5
r14 = r13 + r6
```



Back substituted expression:

r14 = r1*r2+r3+r4-r5+r6

Re-associate and parenthesize to reduce height:

r14 = ((r1*r2)+(((r3+r6)+r4)-r5))

Final assembly code:

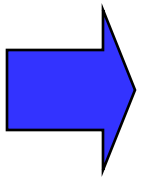
```
t1 = r1 * r2
t2 = r3 + r6
t3 = t2 + r4
t4 = t3-r5
r14 =t1 + t4
```

Back susbstitute
Re-express in tree-height reduced form
Account for latency and arrival times

Optimizing Unrolled Loops

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
       r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

unroll 3 times



```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       ----------
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter2  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       ----------
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter3  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

Unroll = replicate loop body n-1 times.

Hope to enable overlap of operation execution from different iterations

Not possible!

Register Renaming on Unrolled Loop

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       ------------------
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter2  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       ------------------
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter3  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       ------------------
       r11 = load(r2)
       r13 = load(r4)
       r15 = r11 * r13
iter2  r6 = r6 + r15
       r2 = r2 + 4
       r4 = r4 + 4
       ------------------
       r21 = load(r2)
       r23 = load(r4)
       r25 = r21 * r23
iter3  r6 = r6 + r25
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

Register Renaming is Not Enough!

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       ------------------
       r11 = load(r2)
       r13 = load(r4)
       r15 = r11 * r13
iter2  r6 = r6 + r15
       r2 = r2 + 4
       r4 = r4 + 4
       ------------------
       r21 = load(r2)
       r23 = load(r4)
       r25 = r21 * r23
iter3  r6 = r6 + r25
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

- Still not much overlap possible
- Problems
  - r2, r4, r6 sequentialize the iterations
  - Need to rename these
- 2 specialized renaming optis
  - Accumulator variable expansion (r6)
  - Induction variable expansion (r2, r4)

Accumulator Variable Expansion

```
        r16 = r26 = 0
loop:   r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter1   r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
        ------------------
        r11 = load(r2)
        r13 = load(r4)
        r15 = r11 * r13
iter2   r16 = r16 + r15
        r2 = r2 + 4
        r4 = r4 + 4
        ------------------
        r21 = load(r2)
        r23 = load(r4)
        r25 = r21 * r23
iter3   r26 = r26 + r25
        r2 = r2 + 4
        r4 = r4 + 4
        if (r4 < 400) goto loop
        r6 = r6 + r16 + r26
```

- Accumulator variable
  - x = x + y or x = x – y
  - where y is loop variant!!
- Create n-1 temporary accumulators
- Each iteration targets a different accumulator
- Sum up the accumulator variables at the end
- May not be safe for floating-point values

Induction Variable Expansion

```
          r12 = r2 + 4, r22 = r2 + 8
          r14 = r4 + 4, r24 = r4 + 8
          r16 = r26 = 0
loop:     r1 = load(r2)
          r3 = load(r4)
          r5 = r1 * r3
iter1     r6 = r6 + r5
          r2 = r2 + 12
          r4 = r4 + 12
          ------------------
          r11 = load(r12)
          r13 = load(r14)
          r15 = r11 * r13
iter2     r16 = r16 + r15
          r12 = r12 + 12
          r14 = r14 + 12
          ------------------
          r21 = load(r22)
          r23 = load(r24)
          r25 = r21 * r23
iter3     r26 = r26 + r25
          r22 = r22 + 12
          r24 = r24 + 12
          if (r4 < 400) goto loop
          r6 = r6 + r16 + r26
```

- Induction variable
  - x = x + y or x = x – y
  - where y is loop invariant!!
- Create n-1 additional induction variables
- Each iteration uses and modifies a different induction variable
- Initialize induction variables to init, init+step, init+2*step, etc.
- Step increased to n*original step
- Now iterations are completely independent !!

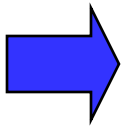
Better Induction Variable Expansion

```
        r16 = r26 = 0
loop:   r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter1   r6 = r6 + r5

        ------------------
        r11 = load(r2+4)
        r13 = load(r4+4)
        r15 = r11 * r13
iter2   r16 = r16 + r15

        ------------------
        r21 = load(r2+8)
        r23 = load(r4+8)
        r25 = r21 * r23
iter3   r26 = r26 + r25
        r2 = r2 + 12
        r4 = r4 + 12
        if (r4 < 400) goto loop
        r6 = r6 + r16 + r26
```

- With base+displacement addressing, often don't need additional induction variables
  - Just change offsets in each iterations to reflect step
  - Change final increments to n * original step

Homework Problem

```
loop:
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400) goto loop
```



```
loop:
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400) goto loop
```

Optimize the unrolled loop

Renaming
Tree height reduction
Ind/Acc expansion

New Topic - Code Generation

- Map optimized “machine-independent” assembly to final assembly code
- Input code
  - Classical optimizations
  - ILP optimizations
  - Formed regions (sbs, hbs), applied if-conversion (if appropriate)
- Virtual → physical binding
  - 2 big steps
  - 1. Scheduling
    - Determine when every operation executions
    - Create MultiOps
  - 2. Register allocation
    - Map virtual → physical registers
    - Spill to memory if necessary

What Do We Need to Schedule Operations?

- Information about the processor
  - Number of resources
  - Which resources are used by each operation
  - Operation latencies
  - Operand encoding limitations
  - For example:
    - 2 issue slots, 1 memory port, 1 adder/multiplier
    - load = 2 cycles, add = 1 cycle, mpy = 3 cycles; all fully pipelined
    - Each operand can be register or 6 bit signed literal
- Ordering constraints amongst operations
  - What order defines correct program execution?
  - Need a precedence graph – flow, anti, output deps
    - What about memory deps? control deps? Delay slots?

How Do We Schedule?

- When is it legal to schedule an instruction?
  - Correct execution is maintained
  - Resources not oversubscribed
- Given multiple operations that can be scheduled, how do you pick the best one?
  - How do you know it is the best one?
    - What about a good guess?
    - Does it matter, just pick one at random?
  - Are decisions final?, or is this an iterative process?
- How do we keep track of resources that are busy/free
  - Need a reservation table
    - Matrix (resources x time)

```
r1 = load(r10)
r2 = load(r11)
r3 = r1 + 4
r4 = r1 – r12
r5 = r2 + r4
r6 = r5 + r3
r7 = load(r13)
r8 = r7 * 23
store (r8, r6)
```

More Stuff to Worry About

- Model more resources
  - Register ports, output busses
  - Non-pipelined resources
- Dependent memory operations
- Multiple clusters
  - Cluster = group of FUs connected to a set of register files such that an FU in a cluster has immediate access to any value produced within the cluster
  - Multicluster = Processor with 2 or more clusters, clusters often interconnected by several low-bandwidth busses
    - Bottom line = Non-uniform access latency to operands
- Scheduler has to be fast
  - NP complete problem
  - So, need a heuristic strategy
- What is better to do first, scheduling or register allocation?

Schedule Before or After Register Allocation?

virtual registers

r1 = load(r10)
r2 = load(r11)
r3 = r1 + 4
r4 = r1 – r12
r5 = r2 + r4
r6 = r5 + r3
r7 = load(r13)
r8 = r7 * 23
store (r8, r6)

physical registers

R1 = load(R1)
R2 = load(R2)
R5 = R1 + 4
R1 = R1 – R3
R2 = R2 + R1
R2 = R2 + R5
R5 = load(R4)
R5 = R5 * 23
store (R5, R2)

Too many artificial ordering constraints if schedule after allocation!!!!

Code Gen: The 6 Step Program

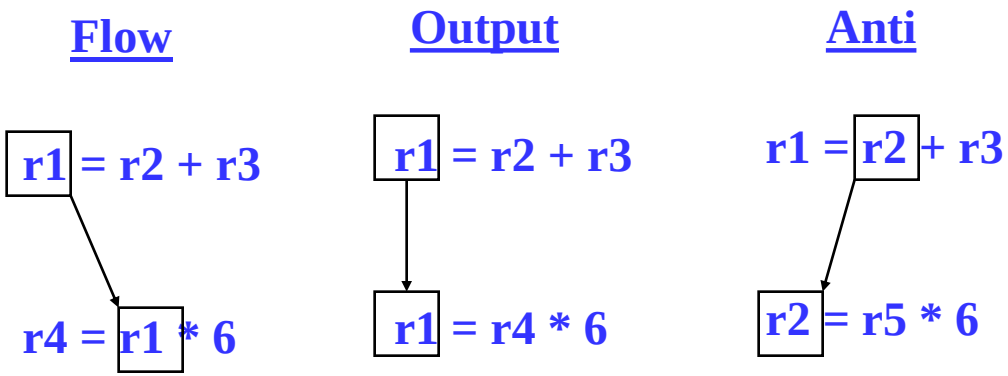
- 1. Code selection, Literal handling
  - Semantic operations to generic operations
  - How to realize a specific function on this machine
  - Complement all bits → xor with –1
  - Can literal be encoded in operation, if not need load/move
- 2. Prepass operation binding
  - Partially bind operation to subset of resources
  - Resources are access equivalent
    - Any choice is equal to any other choice
  - Multi-cluster machine – bind operation to a cluster
- 3. Scheduling
  - What time the operation will be executed
  - What execution resources will be used
    - Chooses alternative

Code Gen: The 6 Step Program (cont)

- 4. Register allocation
  - Assign physical registers
  - Bind each access-equivalent register to a specific physical register
  - Introduce additional code to spill registers to memory
- 5. Postpass scheduling
  - A second pass of scheduling to handle spill code
  - Resource assignments from first pass are ignored
  - But, registers are physical, so less code motion freedom
- 6. Code emission
  - Convert “fully qualified” operations into real assembly
  - A translator basically
  - Assembler converts this assembly to machine code
- **Focus for now on 3, 4, 5, assume 1, 2, 6 are not needed**

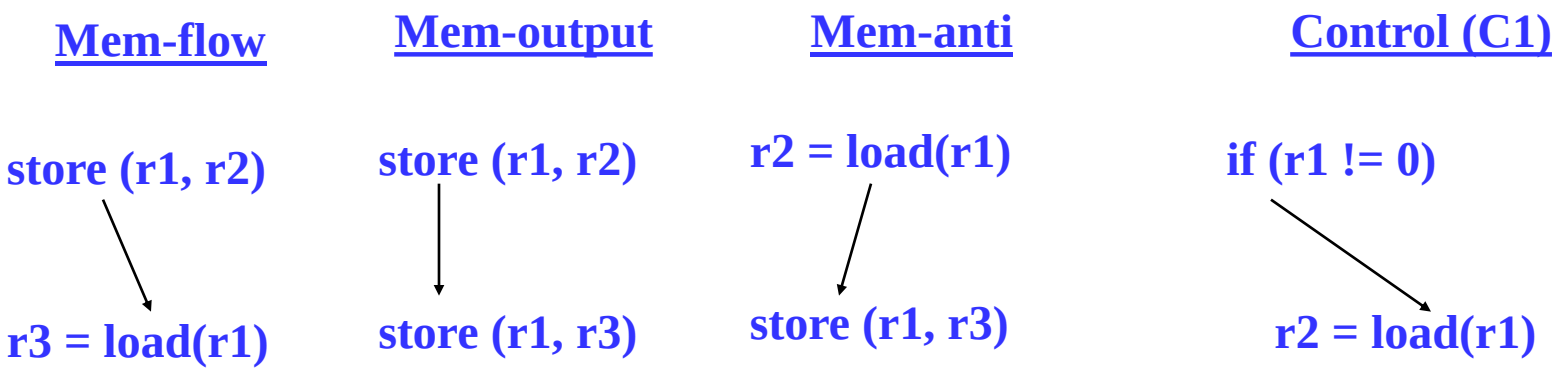
Data Dependences

- 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



More Dependences

- Memory dependences
  - Similar as register, but through memory
  - Memory dependences may be certain or maybe
- Control dependences
  - We discussed this earlier
  - Branch determines whether an operation is executed or not
  - Operation must execute after/before a branch
  - Note, control flow (C0) is not a dependence

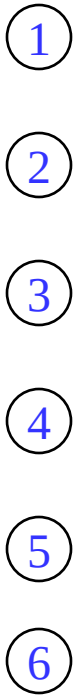


Dependence Graph

- 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) (earliest_read typically 0)
- Register anti dependence, a → b
  - Latest_read(a) – Earliest_write(b) + 1 (latest_read typically equal to earliest_write, so anti deps are 1 cycle)
- Register output dependence, a → b
  - Latest_write(a) – Earliest_write(b) + 1 (earliest_write typically equal to latest_write, so output deps are 1 cycle)
- Negative latency
  - Possible, means successor can start before predecessor
  - We will only deal with latency >= 0, so MAX any latency with 0

Dependence Edge Latencies (2)

- Memory dependences, a → b (all types, flow, anti, output)
  - latency = latest_serialization_latency(a) – earliest_serialization_latency(b) + 1 (generally this is 1)
- 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)
    - conservative, latency = MAX(0, 1-branch_latency)

Class Problem

machine model

latencies

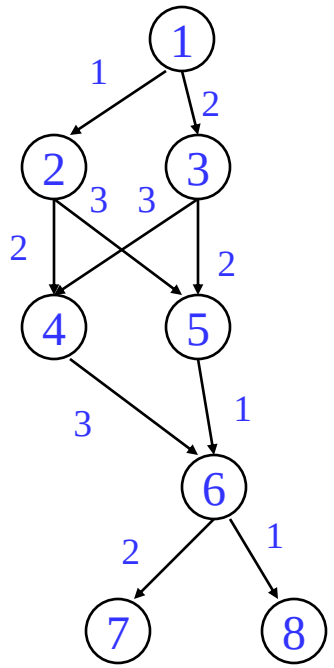
add: 1
mpy: 3
load: 2
sync 1
store: 1
sync 1

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)
```

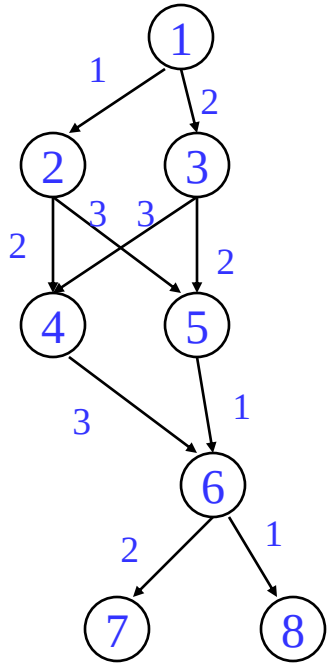
Dependence Graph Properties - Estart

- 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



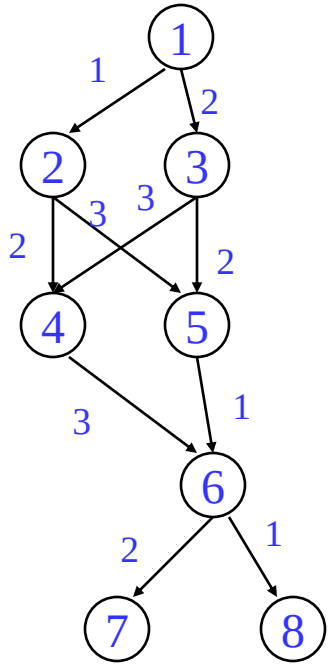
Lstart

- Lstart = latest start time, ALAP
  - Latest time a node can be scheduled s.t. sched length not increased beyond infinite resource schedule length
  - Lstart = Estart if node has no successors
  - Lstart = MIN(Lstart(succ) - latency) for each successor node
  - Example



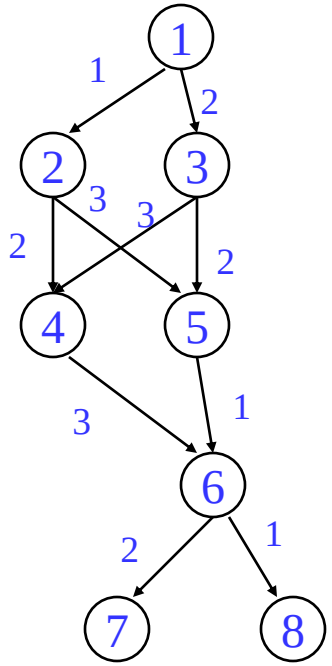
Slack

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

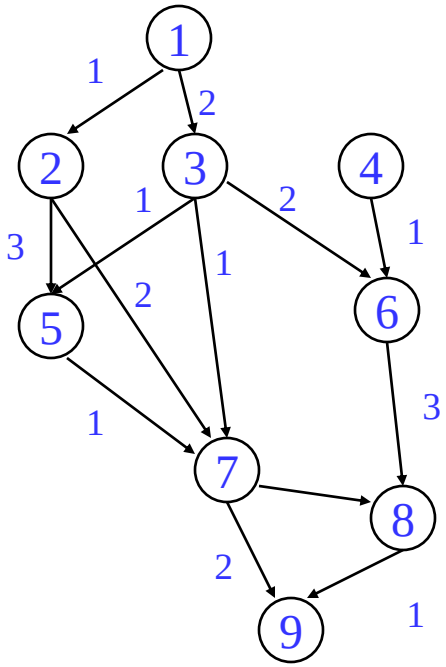


Critical Path

- 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 crit paths



Class Problem



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

Critical path(s) =