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

University of Michigan

September 29, 2025

Announcements & Reading Material

- ❖ Reminder: HW 2
  - » Due next week Wed (Oct 8)
  - » You should have started by now
  - » Talk to GSIs if you are stuck
- ❖ Class project
  - » Focus on project team formation and general topic area
- ❖ 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.
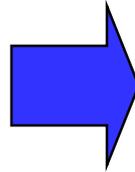
From Last Time: Loop Unrolling

```
for (i=x; i< 100; i++) {
   sum += a[i]*b[i];
}
```

unroll 3 times

```
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 = replicate loop body
n-1 times.

Hope to enable overlap of
operation execution from
different iterations

```
loop:   r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
        r6 = r6 + r5
iter1   r2 = r2 + 4
        r4 = r4 + 4
        if (r4 >= 400) goto exit
        r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter2   r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
        if (r4 >= 400) goto exit
        r1 = load(r2)
        r3 = load(r4)
iter3   r5 = r1 * r3
        r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
        if (r4 < 400) goto loop
exit:
```
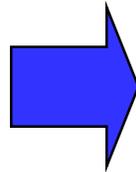
Smarter Loop Unrolling with Known Trip Count

Want to remove early exit branches

Trip count = 400/4 = 100

r4 = 0

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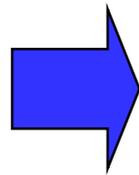   if (r4 < 400) goto loop

unroll multiple
of trip count

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
        ----
        r1 = load(r2)
        r3 = load(r4)
iter4   r5 = r1 * r3
        r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
        if (r4 < 400) goto loop

exit:

What if the Trip Count is not Statically Known?

preloop

```
for (i=0; i< ((400-r4)/4)%3; i++) {
    sum += a[i]*b[i];
}
```

loop:
```
r1 = load(r2)
r3 = load(r4)
```

iter1
```
r5 = r1 * r3
r6 = r6 + r5
r2 = r2 + 4
r4 = r4 + 4
```
- -
```
r1 = load(r2)
r3 = load(r4)
```
iter2
```
r5 = r1 * r3
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
```
exit:

Create a preloop to ensure trip count of unrolled loop is a multiple of the unroll factor

```
r4 = ??
```
loop:
```
r1 = load(r2)
r3 = load(r4)
r5 = r1 * r3
r6 = r6 + r5
r2 = r2 + 4
r4 = r4 + 4
if (r4 < 400) goto loop
```
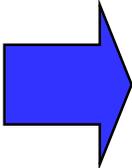
Unrolling Not Enough for Overlapping Iterations: Register Renaming

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 <u>variant</u>!!

❖ 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 <u>invariant</u>!!
- ❖ 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 !!

- 8 -

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
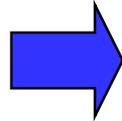
Homework Problem

loop:

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

Optimize the unrolled
loop

Renaming
Tree height reduction
Ind/Acc expansion

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

Homework Problem - Answer

loop:

r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400) goto loop
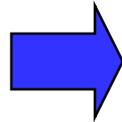
Optimize the unrolled
loop

Renaming
Tree height reduction
Ind/Acc expansion

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

loop:

r1 = load(r2)
r5 = r1 + 3
r6 = r6 + r5
r2 = r2 + 4
r11 = load(r2)
r15 = r11 + 3
r6 = r6 + r15
r2 = r2 + 4
r21 = load(r2)
r25 = r21 + 3
r6 = r6 + r25
r2 = r2 + 4
if (r2 < 400)
    goto loop

after renaming and
tree height reduction

r16 = r26 = 0
loop:

r1 = load(r2)
r5 = r1 + 3
r6 = r6 + r5
r11 = load(r2+4)
r15 = r11 + 3
r16 = r16 + r15
r21 = load(r2+8)
r25 = r21 + 3
r26 = r26 + r25
r2 = r2 + 12
if (r2 < 400)
    goto loop
r6 = r6 + r16
r6 = r6 + r26

after acc and
ind expansion

- 11 -

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 (for VLIW) or reorder instructions (for superscalar)

  » 2. Register allocation

    • Map virtual → physical registers

    • Spill to memory if necessary

Scheduling Operations

❖ Need information about the processor

  » Number of resources, latencies, 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

❖ Need ordering constraints amongst operations

  » What order defines correct program execution?

❖ Given multiple operations that can be scheduled, how do you pick the best one?

  » Is there a best one? Does it matter?

  » Are decisions final?, or is this an iterative process?

❖ How do we keep track of resources that are busy/free

  » Reservation table: Resources x time

Schedule Before or After Register Allocation?

r1 = load(r10)	R1 = load(R1)
r2 = load(r11)	R2 = load(R2)
r3 = r1 + 4	R5 = R1 + 4
r4 = r1 – r12	R1 = R1 – R3
r5 = r2 + r4	R2 = R2 + R1
r6 = r5 + r3	R2 = R2 + R5
r7 = load(r13)	R5 = load(R4)
r8 = r7 * 23	R5 = R5 * 23
store (r8, r6)	store (R5, R2)

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

But, need to schedule after allocation to bind spill code

Solution, do both! Prepass schedule, register allocation, postpass schedule
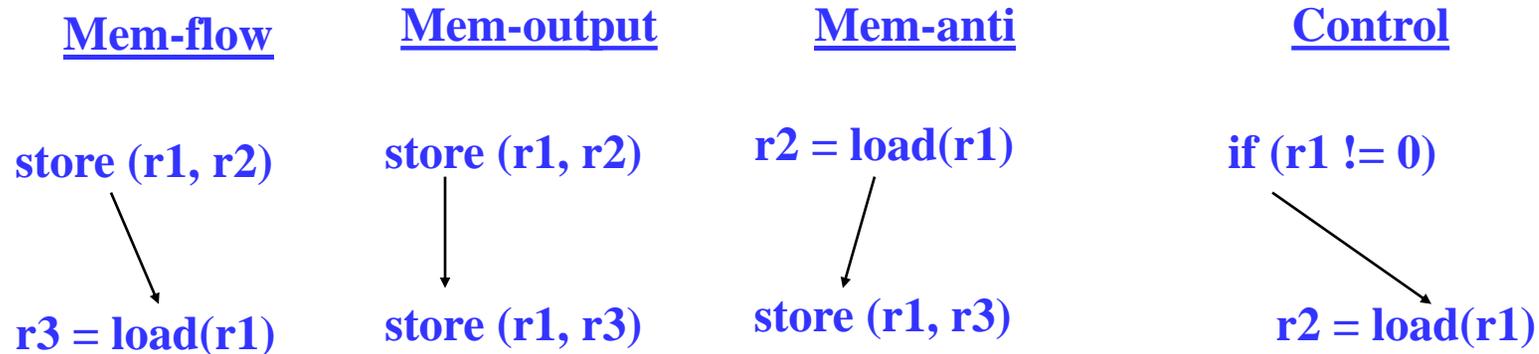
Data Dependences

- ❖ 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 transitively redundant
  - » Types of data dependences

Flow

r1 = r2 + r3

r4 = r1 * 6

Output

r1 = r2 + r3

r1 = r4 * 6

Anti

r1 = r2 + r3

r2 = r5 * 6
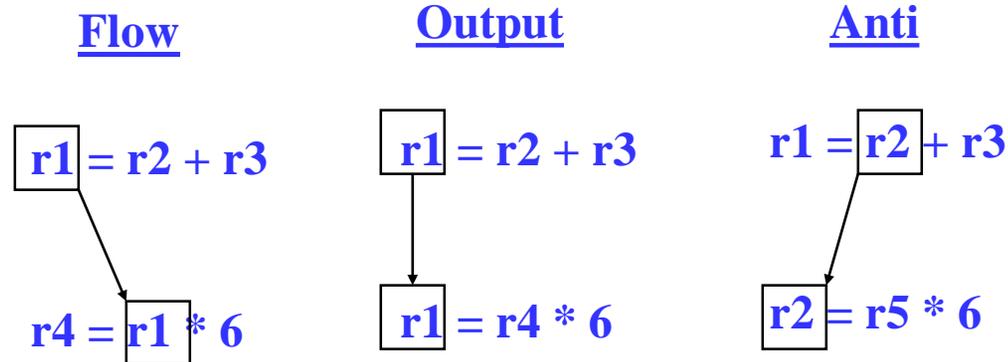
More Dependences

❖ 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

Mem-flow	**Mem-output**	**Mem-anti**	**Control**
store (r1, r2)	store (r1, r2)	r2 = load(r1)	if (r1 != 0)
r3 = load(r1)	store (r1, r3)	store (r1, r3)	r2 = load(r1)

Dependence Graph

❖ Represent dependences between operations in a block via a DAG

    » Nodes = operations/instructions

    » 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)
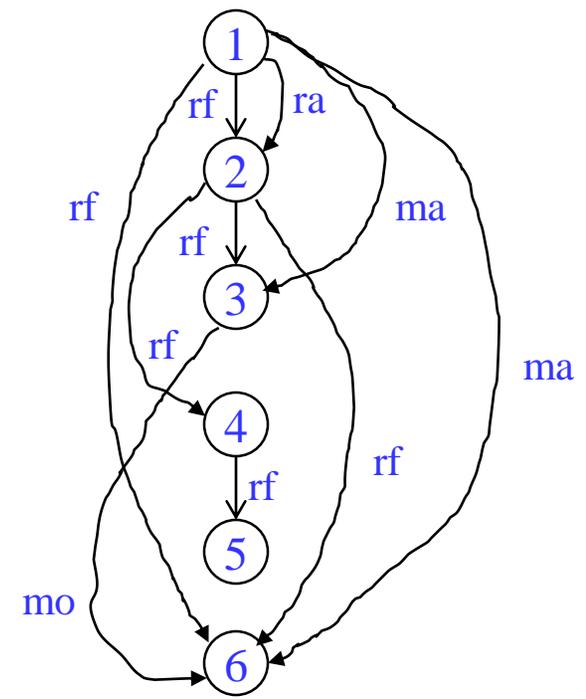
    BB3:

①

②

③

④

⑤

⑥

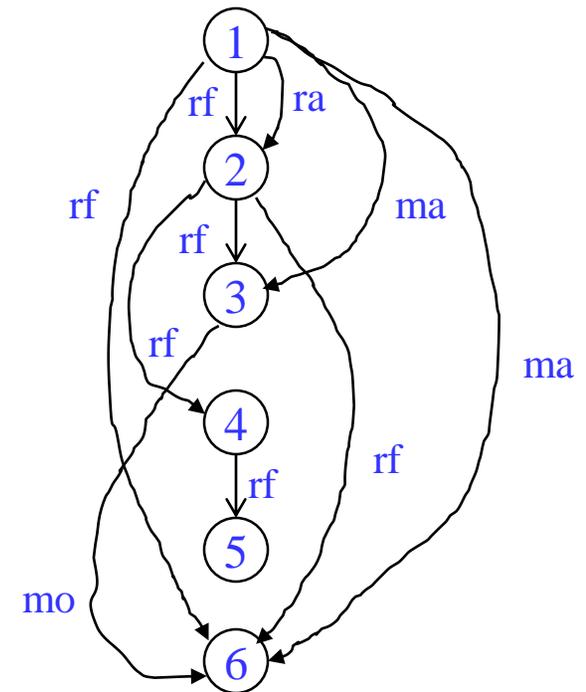# Dependence Graph - Solution

❖ 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:

Instructions 1-4 have control dependence to instruction 5

5→6 control dependence

- 18 -

Dependence Edge Latencies

- ❖ <u>Edge latency</u> = minimum number of cycles necessary between initiation of the predecessor and successor in order to satisfy the dependence
- ❖ Register flow dependence, a = b + c ➔ d = a + 1
  - » Latency of producer instruction for most processors
- ❖ Register anti dependence, a = b + c ➔ b = d + e
  - » 0 cycles for most processors
- ❖ Register output dependence, a = b + c ➔ a = d + e
  - » 1 cycle for most processors
- ❖ Is negative latency possible?
  - » Yes, means successor can start before predecessor
  - » We will only deal with latency >= 0

Dependence Edge Latencies (2)

- ❖ Memory dependences
  - » Store → load (memory flow)
  - » Load → Store (memory anti)
  - » Store → Store (memory output)
  - » All 1 cycle for most processors
- ❖ Control dependences
  - » branch → b
    - Instructions inside then/else paths dependent on branch
    - 1 cycle for most processors
  - » a → branch
    - Op a must be issued before the branch completes
    - 0 cycles for most processors

Class Problem – Add Latencies to Dependence Edges

latencies

add: 1
cmpp: 1
load: 2
store: 1

❖ 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:

Instructions 1-4 have control dependence to instruction 5

5→6 control dependence

Homework Problem 1 – Answer Next Time

machine model

latencies

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

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

1. r1 = load(r2)
2. r2 = r2 + 1
3. store (r8, r2)
4. r3 = load(r2)
5. r4 = r1 * r3
6. r5 = r5 + r4
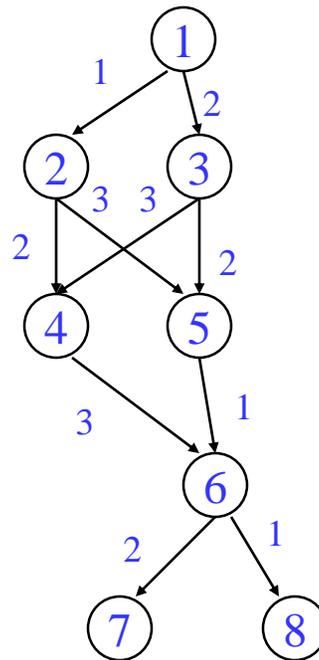7. r2 = r6 + 4
8. store (r2, r5)

① 1

② 2

③ 3

④ 4

⑤ 5

⑥ 6

⑦ 7

⑧ 8

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
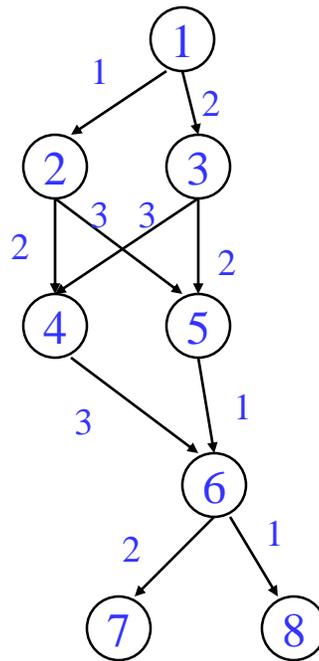
Slack

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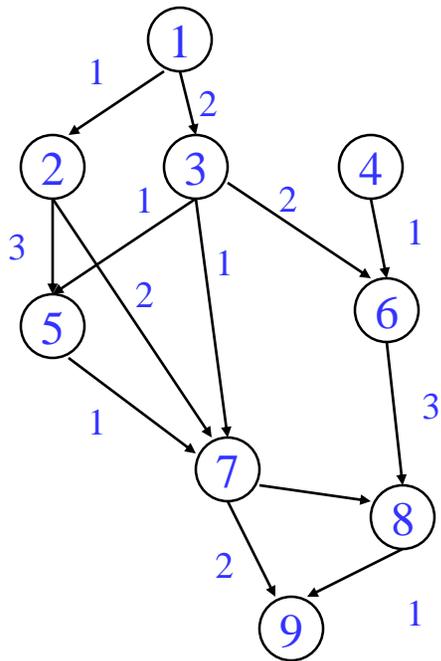 - » Larger slack means more mobility
  - » Example
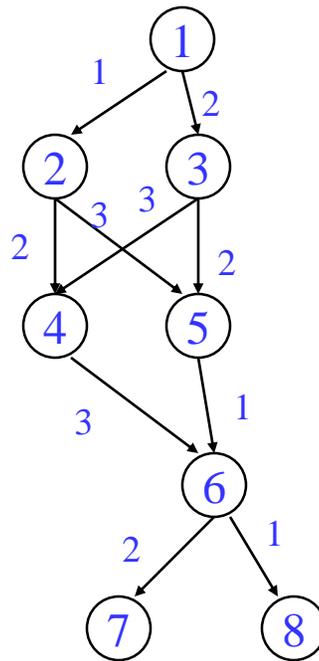
Critical Path

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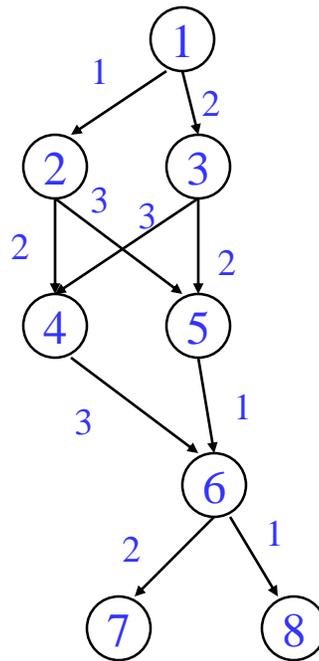
Node	Estart	Lstart	Slack
1			
2			
3			
4			
5			
6			
7			
8			
9			

Critical path(s) =