# EECS 583 – Class 12
# Superblock Scheduling, Intro to Modulo Scheduling

*University of Michigan*

*October 6, 2025*

# Announcements & Reading Material

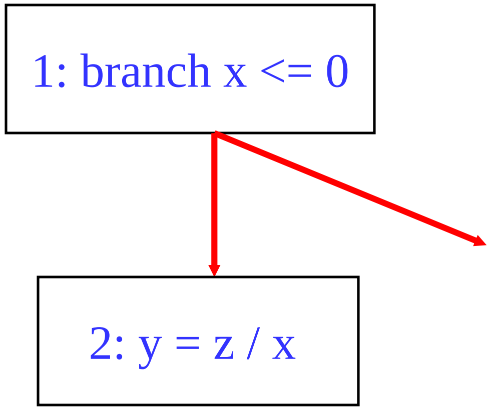
- Homework 2 – Due Wednesday (Oct 8) midnight
- Project discussion meetings – signup next week, meetings week of Oct 20
  - Each group meets 10 mins with Naveen, Rishika, and I
  - Action items
    - Need to identify group members
    - Use piazza to recruit additional group members or express your availability
    - Think about project areas that you want to work on
- Today's class
  - "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops", B. Rau, MICRO-27, 1994, pp. 63-74.
- Next class
  - "Code Generation Schema for Modulo Scheduled Loops", B. Rau, M. Schlansker, and P. Tirumalai, MICRO-25, Dec. 1992.

# Recap: Upward Code Motion Across Branches

- Restriction 1a (register op)
  - The destination of op is not in liveout(br)
  - Wrongly kill a live value
- Restriction 1b (memory op)
  - Op does not modify the memory
  - Actually live memory is what matters, but that is often too hard to determine
- Restriction 2
  - Op must not cause an exception that may terminate the program execution when br is taken
  - Op is executed more often than it is supposed to (speculated)
  - Page fault or cache miss are ok
- Insert control dep when either restriction is violated

```
…
if (x > 0)
   y = z / x
…
```

control flow graph



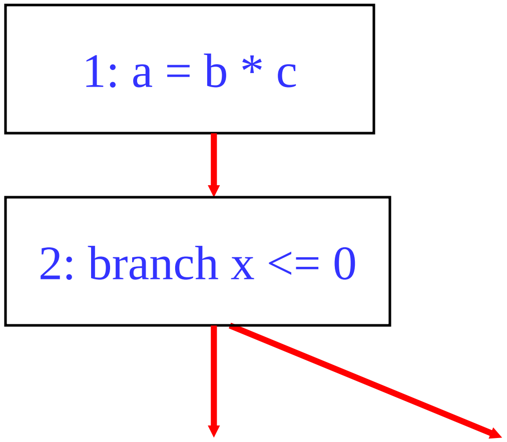
# Recap: Downward Code Motion Across Branches

- Restriction 1 (liveness)
  - » If no compensation code
    - Same restriction as before, destination of op is not liveout
  - » Else, no restrictions
    - Duplicate operation along both directions of branch if destination is liveout
- Restriction 2 (speculation)
  - » Not applicable, downward motion is not speculation
- Again, insert control dep when the restrictions are violated
- Part of the philosphy of superblocks is no compensation code insertion hence R1 is enforced!

```
…
a = b * c
if (x > 0)

else
…
```

control flow graph

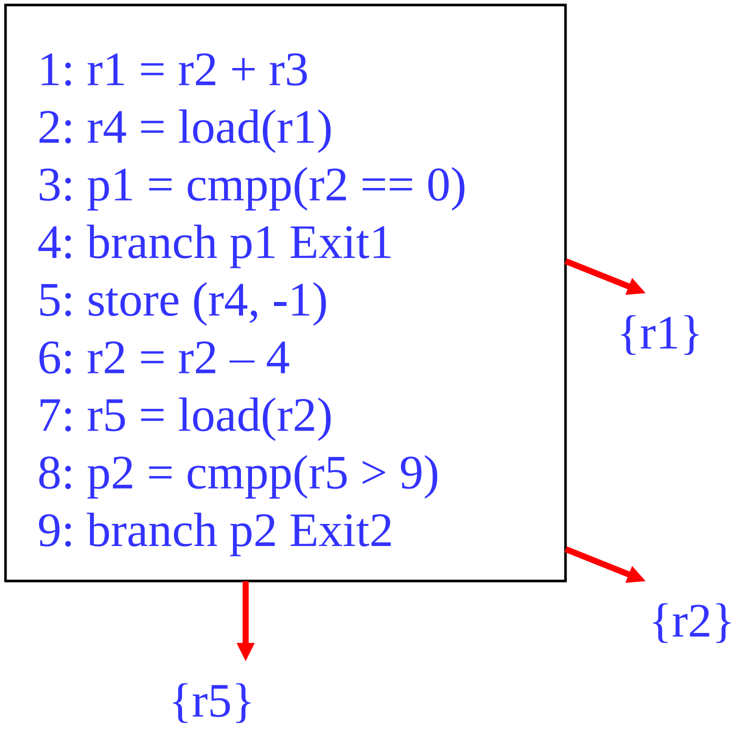

1: a = b * c

2: branch x <= 0

# Add Control Dependences to a Superblock

Superblock

Assumed liveout sets

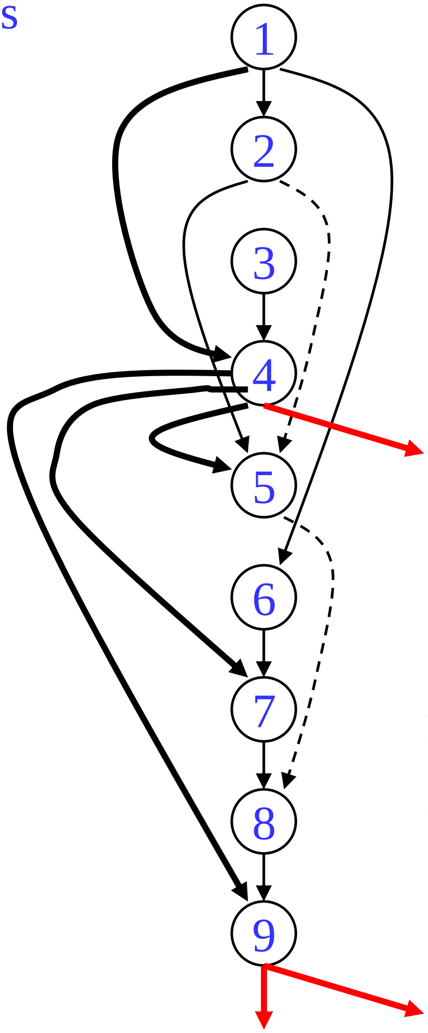


```
1: r1 = r2 + r3
2: r4 = load(r1)
3: p1 = cmpp(r2 == 0)
4: branch p1 Exit1
5: store (r4, -1)
6: r2 = r2 – 4
7: r5 = load(r2)
8: p2 = cmpp(r5 > 9)
9: branch p2 Exit2
```

{r1}

{r2}

{r5}



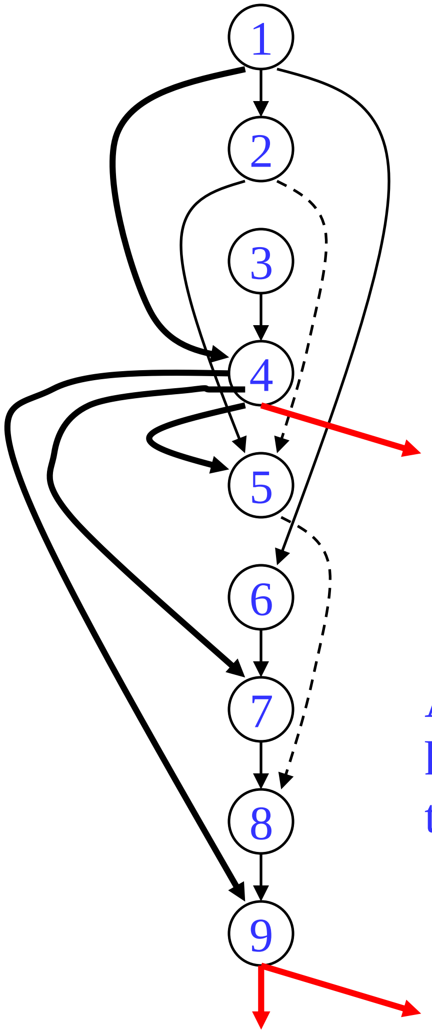
Dependences
Bold edges are control
Dashed edges are memory
Non-bold are register

All ops have cdep to op 9!

Notes: All branches are control dependent on one another.
If no compensation, all ops dependent on last branch

# List Scheduling on Superblocks

- Follow same algorithm as BBs
- Steps
  - Draw data dependence graph
  - Compute Estart, all Lstarts, priority
  - Perform list scheduling
- Scheduling process
  - Ignore side exits – treat SB just like a BB
  - Control dependences prevent illegal code motion across branches

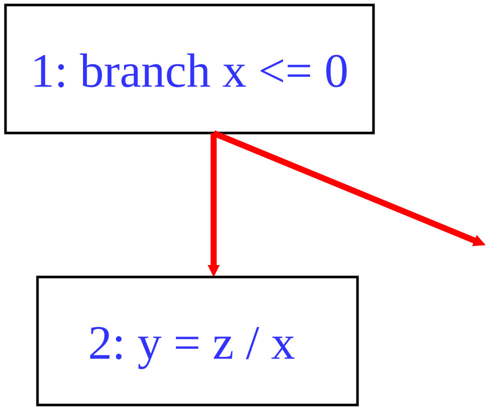


Dependences
Bold edges are control
Dashed edges are memory
Non-bold are register

All ops have cdep to op 9!

# Relaxing Code Motion Restrictions

- Upward code motion is generally more effective
  - Speculate that an op is useful (just like an out-of-order processor with branch pred)
  - Start ops early, hide latency, overlap execution, more parallelism
- Removing restriction 1
  - For register ops – use register renaming
  - Could rename memory too, but generally not worth it
- Removing restriction 2
  - Restricted speculation: Some ops don’t cause exceptions – This is what we have been doing! (load, store, int divide, FP can cause exceptions)
  - Need hardware support (aka <u>speculation models</u>) to speculate potentially excepting instructions
    - Ignore exceptions
    - Delay exceptions



R1: y is not in liveout(1)

R2: op 2 will never cause an exception when op1 is taken

# General Speculation Model

- 2 types of exceptions
  - Program terminating (traps)
    - Div by 0, illegal address
  - Fixable (normal and handled at run time)
    - Page fault, TLB miss
- General speculation
  - Processor provides non-trapping versions of all operations (div, load, etc)
  - Return some bogus value (0) when error occurs
  - R2 is completely ignored, only R1 limits speculation
  - Speculative ops converted into non-trapping version
  - Fixable exceptions handled as usual for non-trapping ops

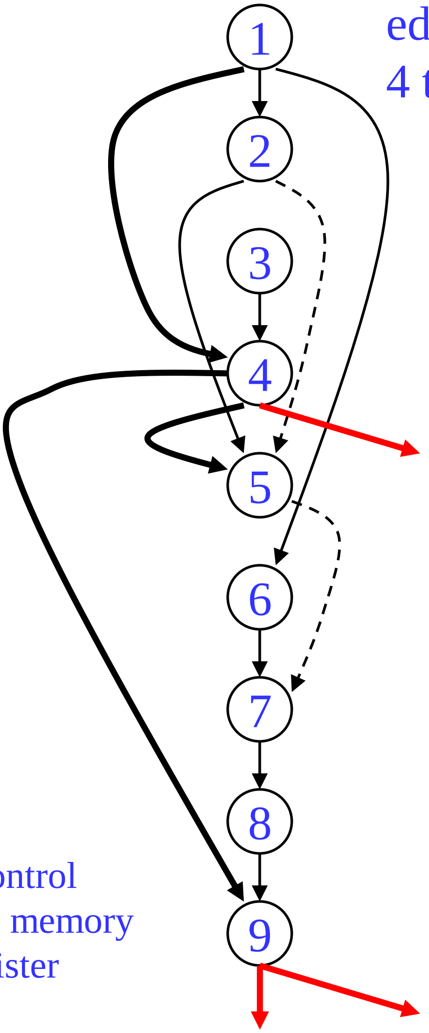
```
1: r1 = r2 + r3
2: r4 = load(r1)
3: p1 = cmpp(r2 == 0)
4: branch p1 Exit1
5: store (r4, -1)
6: r2 = r2 – 4
7: r5 = load(r2)
8: p2 = cmpp(r5 > 9)
9: branch p2 Exit2
```

{r1}

{r2}

{r5}

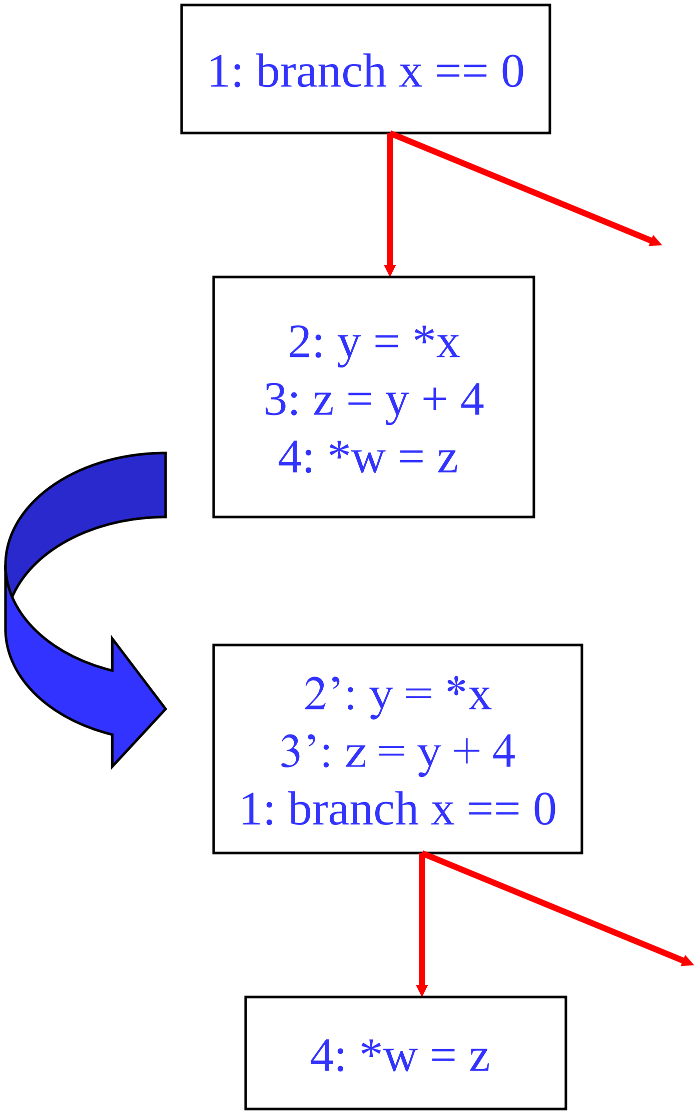
Remove edge from 4 to 7



Dependences
Bold edges are control
Dashed edges are memory
Non-bold are register

# Programming Implications of General Spec

- Correct program
  - No problem at all
  - Exceptions will only result when branch is taken
  - Results of excepting speculative operation(s) will not be used for anything useful (R1 guarantees this!)
- Program debugging
  - Non-trapping ops make this almost impossible
  - Disable general speculation during program debug phase



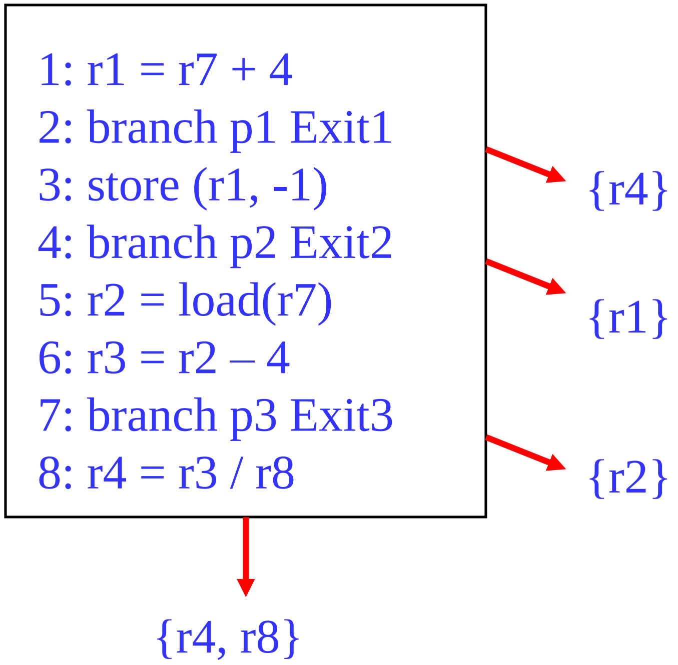
1: branch x == 0

2: y = *x
3: z = y + 4
4: *w = z

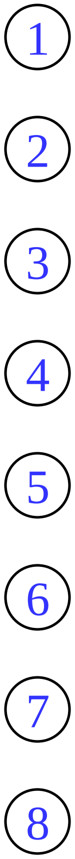
2': y = *x
3': z = y + 4
1: branch x == 0

4: *w = z

# Homework Problem

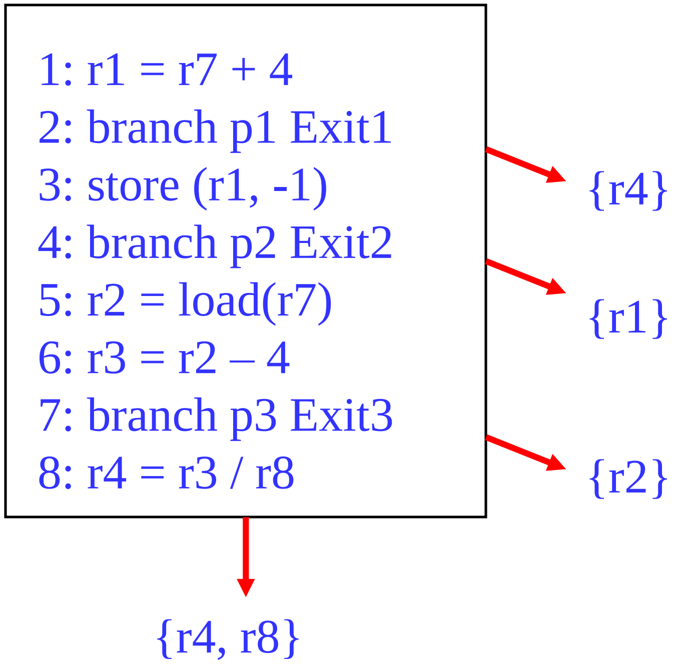


```
1: r1 = r7 + 4
2: branch p1 Exit1
3: store (r1, -1)
4: branch p2 Exit2
5: r2 = load(r7)
6: r3 = r2 – 4
7: branch p3 Exit3
8: r4 = r3 / r8
```



1. Draw the dep graph assuming restricted speculation
2. What edges can be removed if general speculation support is provided?
3. With more renaming, what dependences could be removed?

# Homework Problem – Solution



1: r1 = r7 + 4
2: branch p1 Exit1
3: store (r1, -1)
4: branch p2 Exit2
5: r2 = load(r7)
6: r3 = r2 – 4
7: branch p3 Exit3
8: r4 = r3 / r8

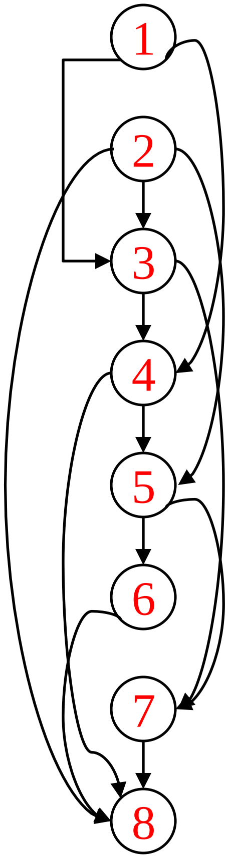
{r4}

{r1}

{r2}

{r4, r8}

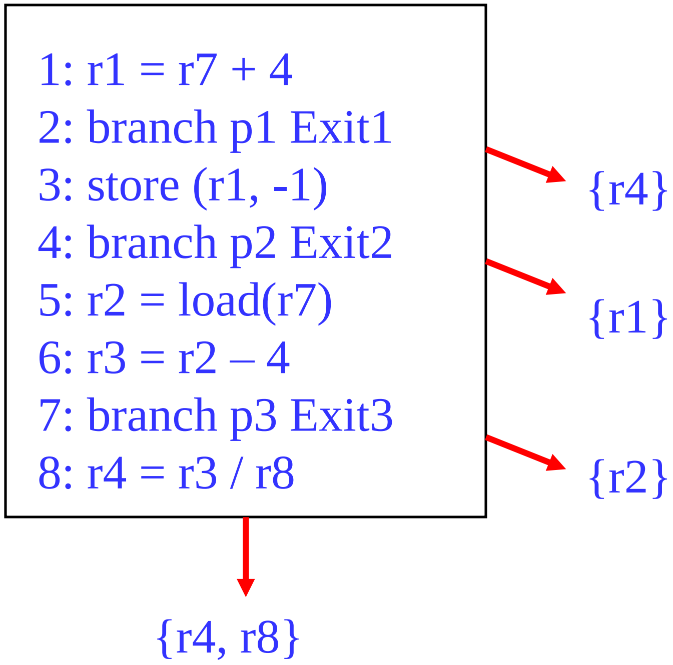
1. Dependence graph with restricted speculation



1. Draw the dep graph assuming restricted speculation
2. What edges can be removed if general speculation support is provided?
3. With more renaming, what dependences could be removed?

Additional control deps: 2→4, 2→7, 4→7
No memory dependence between 3 and 5 since can prove the addresses are always 4 apart

# Homework Problem – Solution (continued)



```
1: r1 = r7 + 4
2: branch p1 Exit1
3: store (r1, -1)
4: branch p2 Exit2
5: r2 = load(r7)
6: r3 = r2 – 4
7: branch p3 Exit3
8: r4 = r3 / r8
```

{r4}

{r1}

{r2}

{r4, r8}

1. Draw the dep graph assuming restricted speculation
2. What edges can be removed if general speculation support is provided?
3. With more renaming, what dependences could be removed?

2. With general speculation, edges from 2→5, 4→5, 4→8, 7→8 can be removed

3. With further renaming, the edge from 2→8 can be removed.

Note, the edge from 2→3 cannot be removed since we conservatively do not allow stores to speculate.

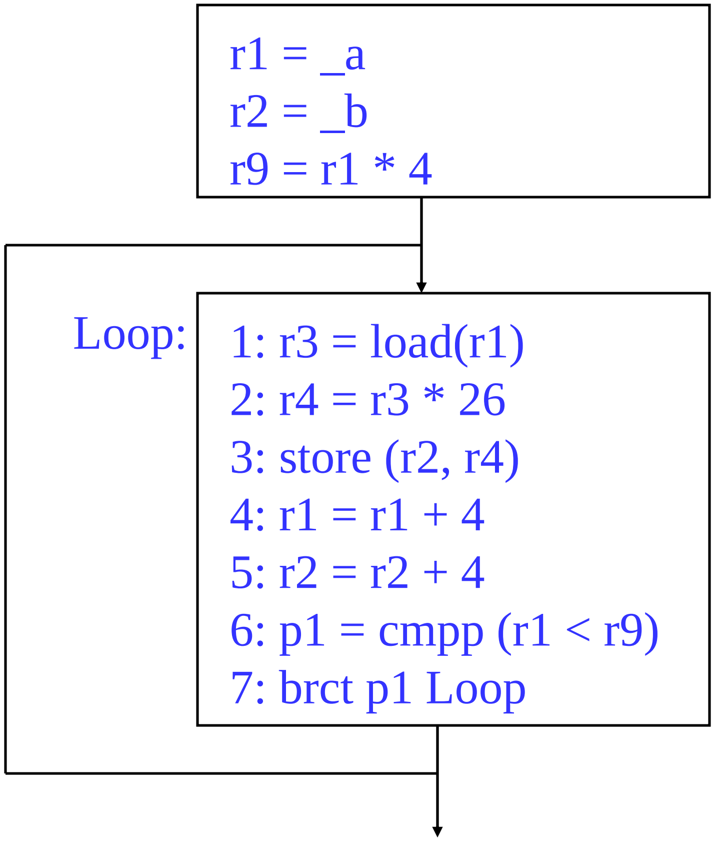
Note2, you do not need general speculation to remove edges from 2→6 and 4→6 since integer subtract never causes exception.

# Change Focus to Scheduling Loops

Most of program execution time is spent in loops

Problem:  How do we achieve compact schedules for loops

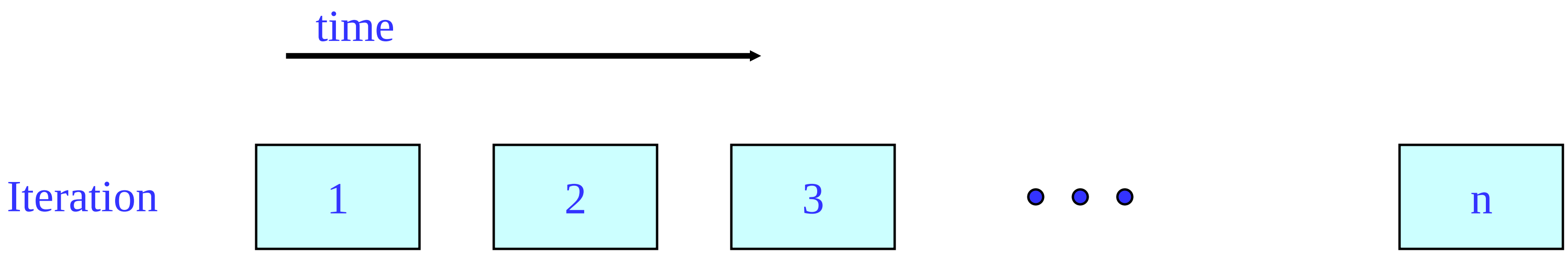
```
for (j=0; j<100; j++)
   b[j] = a[j] * 26
```



```
r1 = _a
r2 = _b
r9 = r1 * 4
```

```
Loop:  1: r3 = load(r1)
       2: r4 = r3 * 26
       3: store (r2, r4)
       4: r1 = r1 + 4
       5: r2 = r2 + 4
       6: p1 = cmpp (r1 < r9)
       7: brct p1 Loop
```

# Basic Approach – List Schedule the Loop Body



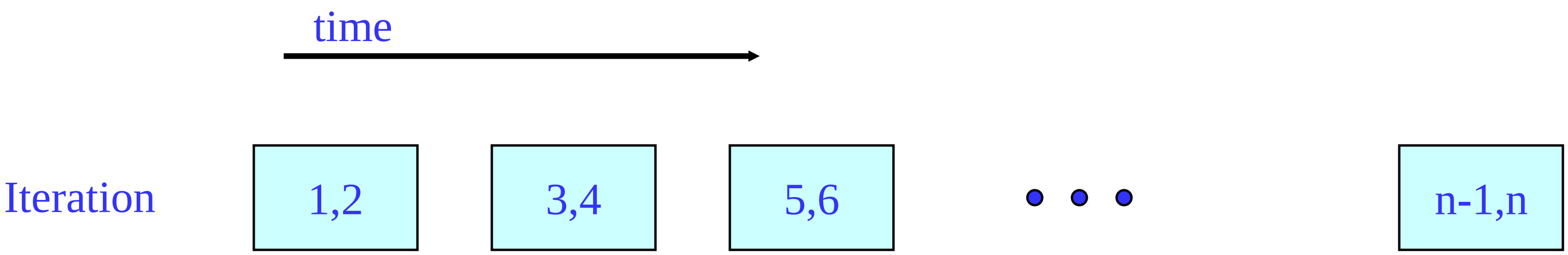
Schedule each iteration
resources: 4 issue, 2 alu, 1 mem, 1 br
latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

```
1: r3 = load(r1)
2: r4 = r3 * 26
3: store (r2, r4)
4: r1 = r1 + 4
5: r2 = r2 + 4
6: p1 = cmpp (r1 < r9)
7: brct p1 Loop
```

| time | ops |
|---|---|
| 0 | 1, 4 |
| 1 | 6 |
| 2 | 2 |
| 3 | - |
| 4 | - |
| 5 | 3, 5, 7 |

Total time = 6 * n

# Unroll Then Schedule Larger Body



time

Iteration

1,2

3,4

5,6

• • •

n-1,n

Schedule each iteration
resources: 4 issue, 2 alu, 1 mem, 1 br
latencies: add=1, cmpp = 1, mpy=3, ld = 2, st = 1, br = 1

```
1: r3 = load(r1)
2: r4 = r3 * 26
3: store (r2, r4)
4: r1 = r1 + 4
5: r2 = r2 + 4
6: p1 = cmpp (r1 < r9)
7: brct p1 Loop
```

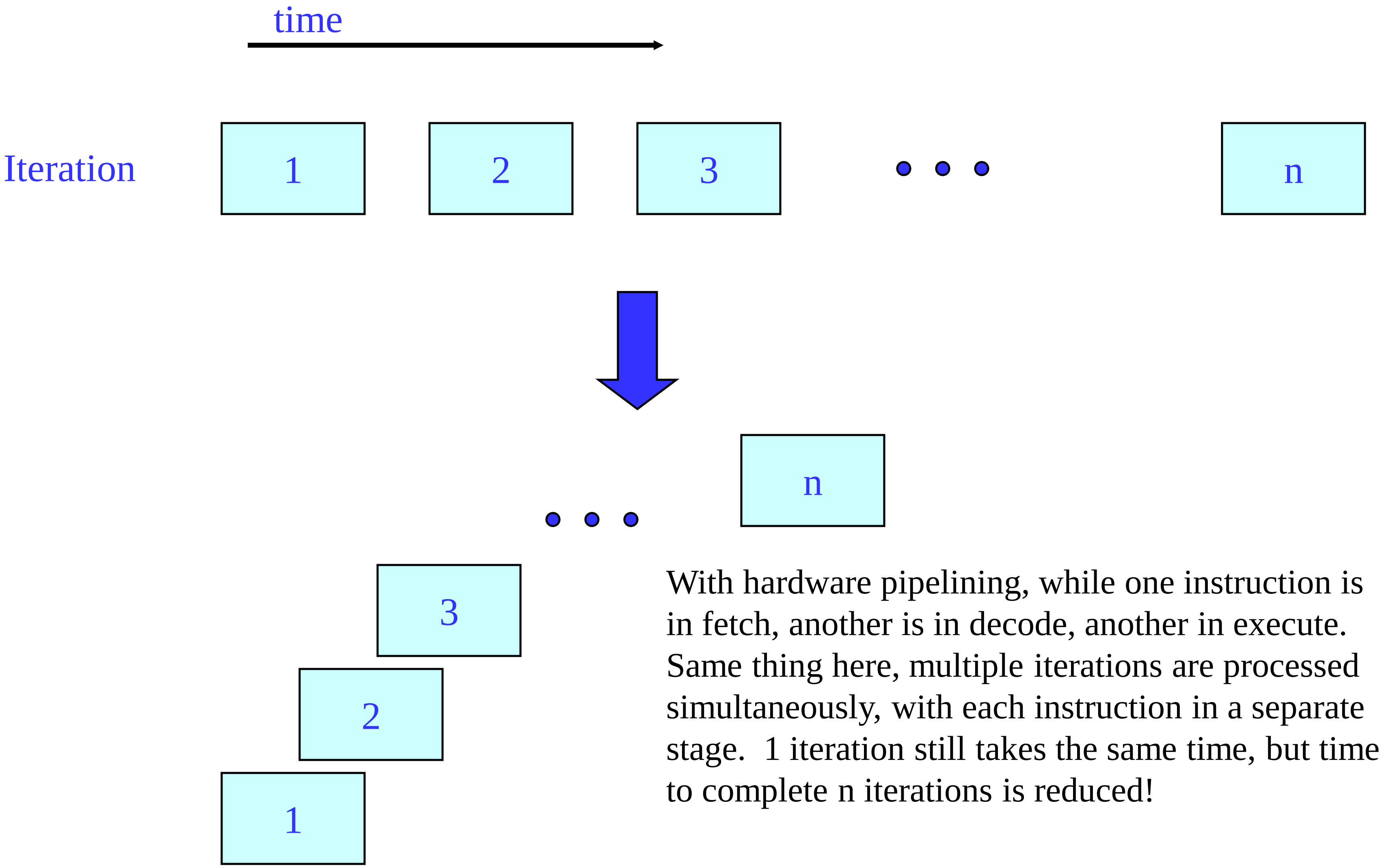
| time | ops |
|---|---|
| 0 | 1, 4 |
| 1 | 1’, 6, 4’ |
| 2 | 2, 6’ |
| 3 | 2’ |
| 4 | - |
| 5 | 3, 5, 7 |
| 6 | 3’,5’,7’ |

Total time = 7 * n/2

# Problems With Unrolling

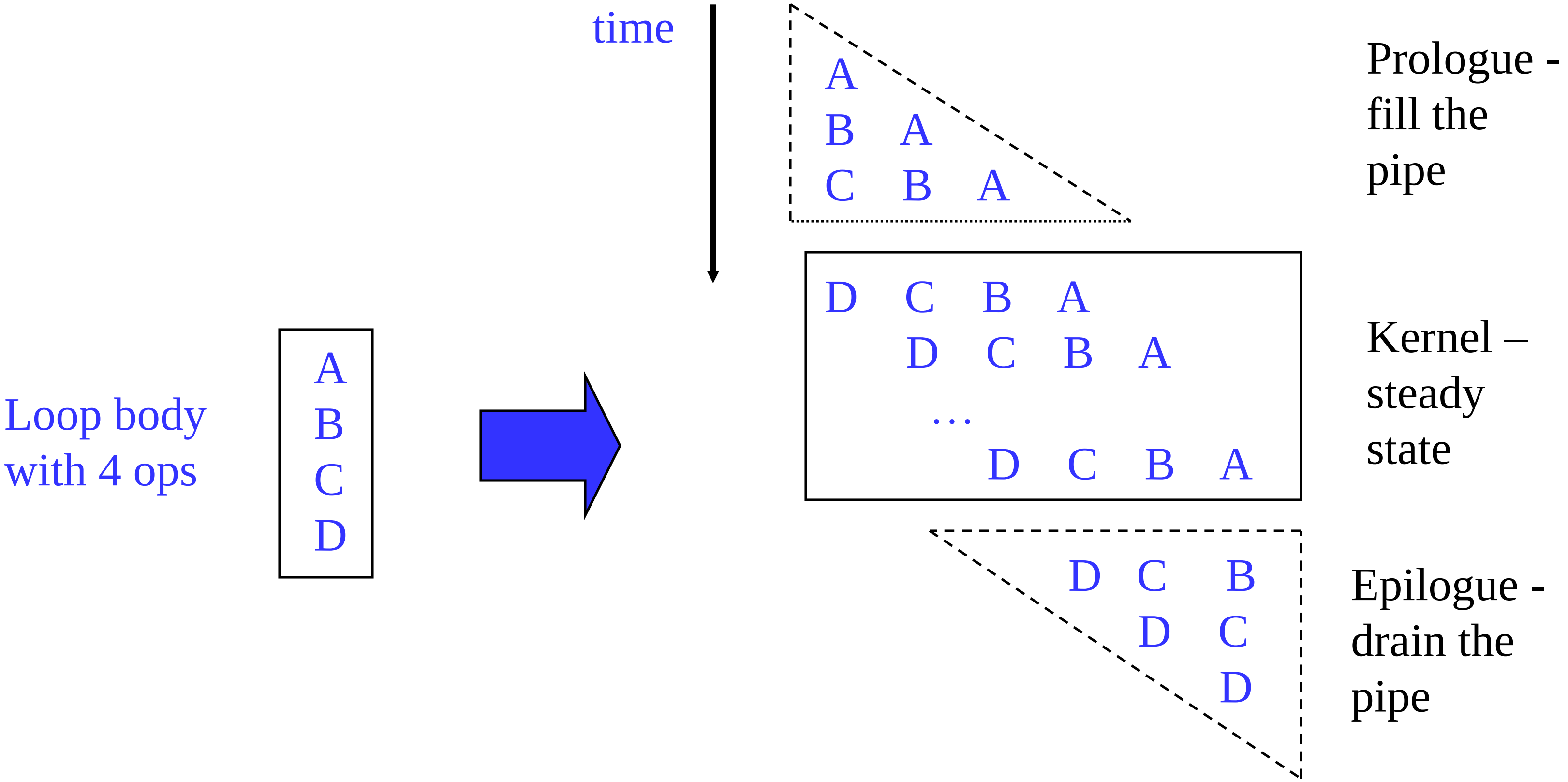
- Code bloat
  - Typical unroll is 4-16x
  - Use profile statistics to only unroll "important" loops
  - But still, code grows fast
- Barrier after across unrolled bodies
  - I.e., for unroll 2, can only overlap iterations 1 and 2, 3 and 4, …
- Does this mean unrolling is bad?
  - No, in some settings its very useful
    - Low trip count
    - Lots of branches in the loop body
  - But, in other settings, there is room for improvement

# Overlap Iterations Using Pipelining



With hardware pipelining, while one instruction is in fetch, another is in decode, another in execute. Same thing here, multiple iterations are processed simultaneously, with each instruction in a separate stage.  1 iteration still takes the same time, but time to complete n iterations is reduced!

# A Software Pipeline



time

A<br>
B A<br>
C B A

Prologue - fill the pipe

Loop body with 4 ops

A<br>
B<br>
C<br>
D

D C B A<br>
D C B A<br>
…<br>
D C B A

Kernel – steady state

D C B<br>
D C<br>
D

Epilogue - drain the pipe

Steady state: 4 iterations executed simultaneously, 1 operation from each iteration. Every cycle, an iteration starts and finishes when the pipe is full.

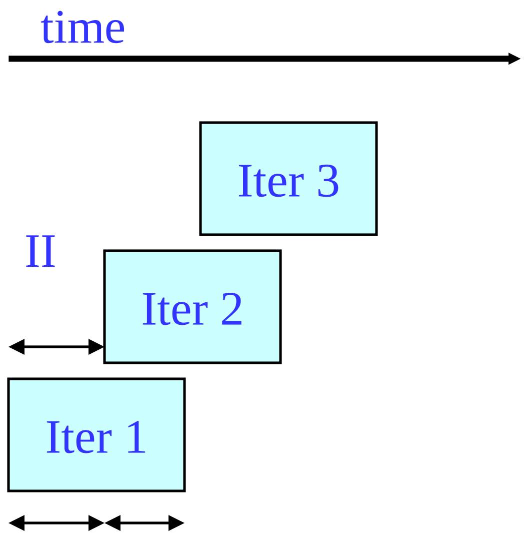
# Creating Software Pipelines

- Lots of software pipelining techniques out there
- Modulo scheduling
  - Most widely adopted
  - Practical to implement, yields good results
- Conceptual strategy
  - Unroll the loop completely
  - Then, schedule the code completely with 2 constraints
    - All iteration bodies have identical schedules
    - Each iteration is scheduled to start some fixed number of cycles later than the previous iteration
  - Initiation Interval (II) = fixed delay between the start of successive iterations
  - Given the 2 constraints, the unrolled schedule is repetitive (kernel) except the portion at the beginning (prologue) and end (epilogue)
    - Kernel can be re-rolled to yield a new loop

# Creating Software Pipelines (2)

- Create a schedule for 1 iteration of the loop such that when the same schedule is repeated at intervals of II cycles
  - No intra-iteration dependence is violated
  - No inter-iteration dependence is violated
  - No resource conflict arises between operation in same or distinct iterations
- We will start out assuming Intel Itanium-style hardware support, then remove it later
  - Rotating registers
  - Predicates
  - Software pipeline loop branch

# Terminology



Initiation Interval (II) = fixed delay between the start of successive iterations

Each iteration can be divided into stages consisting of II cycles each

Number of stages in 1 iteration is termed the stage count (SC)

Takes SC-1 cycles to fill/drain the pipe

# Resource Usage Legality

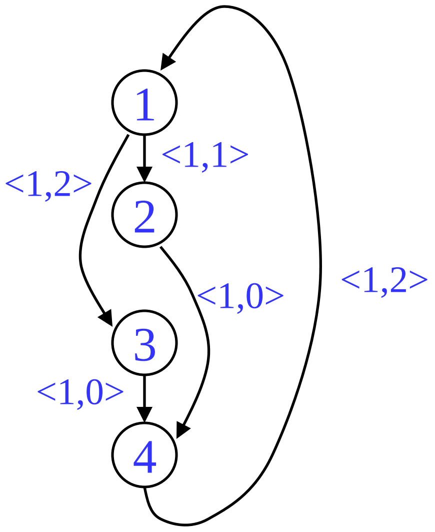
- Need to guarantee that
  - No resource is used at 2 points in time that are separated by an interval which is a multiple of II
  - I.E., within a single iteration, the same resource is never used more than 1x at the same time modulo II
  - Known as modulo constraint, where the name modulo scheduling comes from
  - Modulo reservation table solves this problem
    - To schedule an op at time T needing resource R
      - The entry for R at T mod II must be free
    - Mark busy at T mod II if schedule

II = 3

|   | alu1 | alu2 | mem | bus0 | bus1 | br |
|---|---|---|---|---|---|---|
| 0 |  |  |  |  |  |  |
| 1 |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |

# Dependences in a Loop

- Need worry about 2 kinds
  - Intra-iteration
  - Inter-iteration
- Delay
  - Minimum time interval between the start of operations
  - Operation read/write times
- Distance
  - Number of iterations separating the 2 operations involved
  - Distance of 0 means intra-iteration
- Recurrence manifests itself as a circuit in the dependence graph



Edges annotated with tuple

<delay, distance>

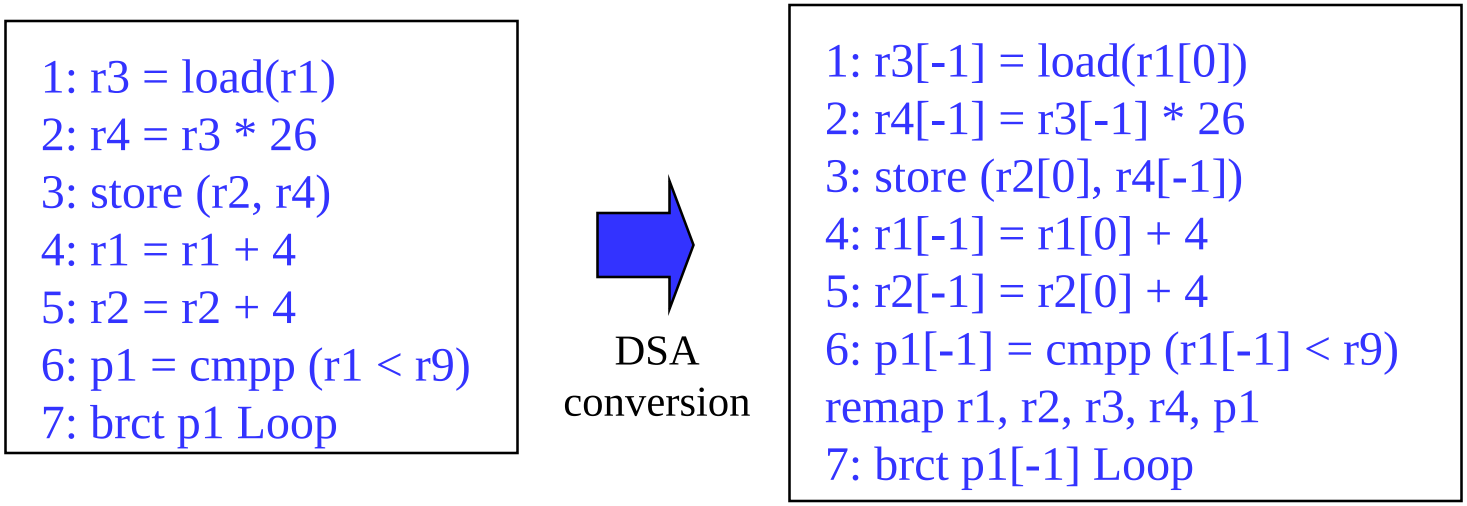
# Dynamic Single Assignment (DSA) Form

Impossible to overlap iterations because each iteration writes to the same register.  So, we’ll have to remove the anti and output dependences.

Virtual rotating registers

- * Each register is an infinite push down array (Expanded virtual reg or EVR)
- * Write to top element, but can reference any element
- * Remap operation slides everything down → r[n] changes to r[n+1]

A program is in DSA form if the same virtual register (EVR element) is never assigned to more than 1x on any dynamic execution path



```
1: r3 = load(r1)
2: r4 = r3 * 26
3: store (r2, r4)
4: r1 = r1 + 4
5: r2 = r2 + 4
6: p1 = cmpp (r1 < r9)
7: brct p1 Loop
```

DSA conversion

```
1: r3[-1] = load(r1[0])
2: r4[-1] = r3[-1] * 26
3: store (r2[0], r4[-1])
4: r1[-1] = r1[0] + 4
5: r2[-1] = r2[0] + 4
6: p1[-1] = cmpp (r1[-1] < r9)
remap r1, r2, r3, r4, p1
7: brct p1[-1] Loop
```

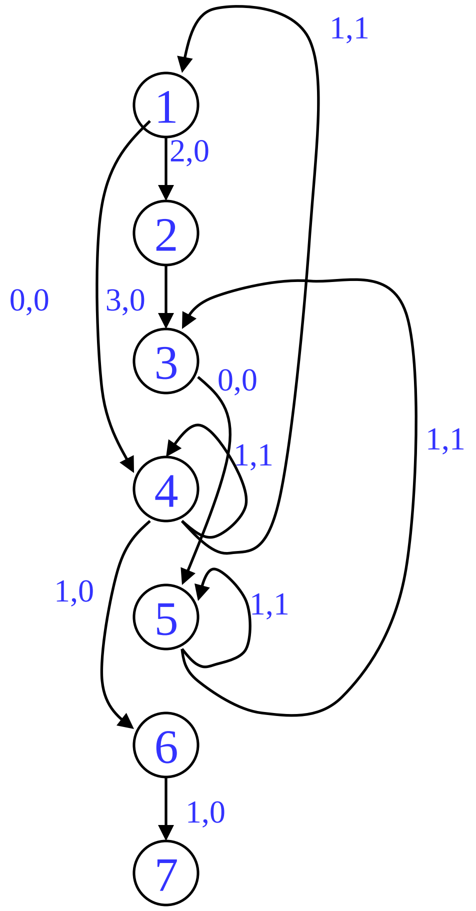
# Physical Realization of EVRs

- EVR may contain an unlimited number values
  - But, only a finite contiguous set of elements of an EVR are ever live at any point in time
  - These must be given physical registers
- Conventional register file
  - Remaps are essentially copies, so each EVR is realized by a set of physical registers and copies are inserted
- Rotating registers
  - Direct support for EVRs
  - No copies needed
  - File “rotated” after each loop iteration is completed

# Loop Dependence Example

```
1: r3[-1] = load(r1[0])
2: r4[-1] = r3[-1] * 26
3: store (r2[0], r4[-1])
4: r1[-1] = r1[0] + 4
5: r2[-1] = r2[0] + 4
6: p1[-1] = cmpp (r1[-1] < r9)
remap r1, r2, r3, r4, p1
7: brct p1[-1] Loop
```

In DSA form, there are no inter-iteration anti or output dependences!

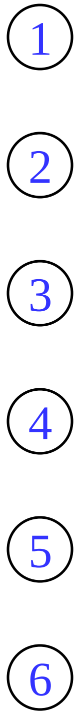


<delay, distance>

# Class Problem

Latencies: ld = 2, st = 1, add = 1, cmpp = 1, br = 1

```
1: r1[-1] = load(r2[0])
2: r3[-1] = r1[1] – r1[2]
3: store (r3[-1], r2[0])
4: r2[-1] = r2[0] + 4
5: p1[-1] = cmpp (r2[-1] < 100)
remap r1, r2, r3
6: brct p1[-1] Loop
```



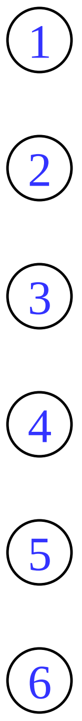
Draw the dependence graph showing both intra and inter iteration dependences

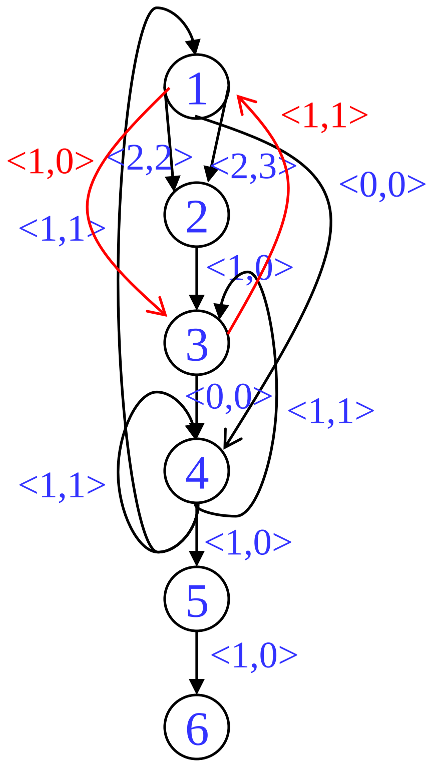
# Class Problem Answer

Latencies: ld = 2, st = 1, add = 1, cmpp = 1, br = 1

```
1: r1[-1] = load(r2[0])
2: r3[-1] = r1[1] – r1[2]
3: store (r3[-1], r2[0])
4: r2[-1] = r2[0] + 4
5: p1[-1] = cmpp (r2[-1] < 100)
remap r1, r2, r3
6: brct p1[-1] Loop
```

Draw the dependence graph showing both intra and inter iteration dependences





Red edges are memory anti and flow dependences

Instructions 1-5 have <0,0> control dependences to 6.

# Minimum Initiation Interval (MII)

- Remember, II = number of cycles between the start of successive iterations
- Modulo scheduling requires a candidate II be selected before scheduling is attempted
  - Try candidate II, see if it works
  - If not, increase by 1, try again repeating until successful
- MII is a lower bound on the II
  - MII = Max(ResMII, RecMII)
  - ResMII = resource constrained MII
    - Resource usage requirements of 1 iteration
  - RecMII = recurrence constrained MII
    - Latency of the circuits in the dependence graph

# ResMII

Concept: If there were no dependences between the operations, what is the the shortest possible schedule?

Simple resource model

A processor has a set of resources R. For each resource r in R there is count(r) specifying the number of identical copies

$$\text{ResMII} = \max_{\text{for all } r \text{ in } R} \left(\text{uses}(r) / \text{count}(r)\right)$$

uses(r) = number of times the resource is used in 1 iteration

In reality its more complex than this because operations can have multiple alternatives (different choices for resources it could be assigned to), but we will ignore this for now

# ResMII Example

ResMII = MAX (uses(r) / count(r))

uses(r) = number of times the resource is used in 1 iteration

resources: 4 issue, 2 alu, 1 mem, 1 br
latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

```
1: r3 = load(r1)
2: r4 = r3 * 26
3: store (r2, r4)
4: r1 = r1 + 4
5: r2 = r2 + 4
6: p1 = cmpp (r1 < r9)
7: brct p1 Loop
```

ALU: used by 2, 4, 5, 6
→ 4 ops / 2 units = 2

Mem: used by 1, 3
→ 2 ops / 1 unit = 2

Br: used by 7
→ 1 op / 1 unit = 1

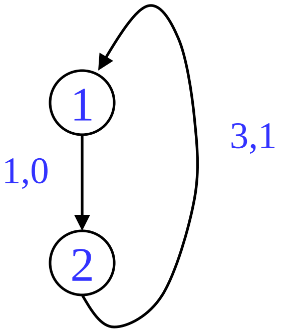
ResMII = MAX(2,2,1) = 2

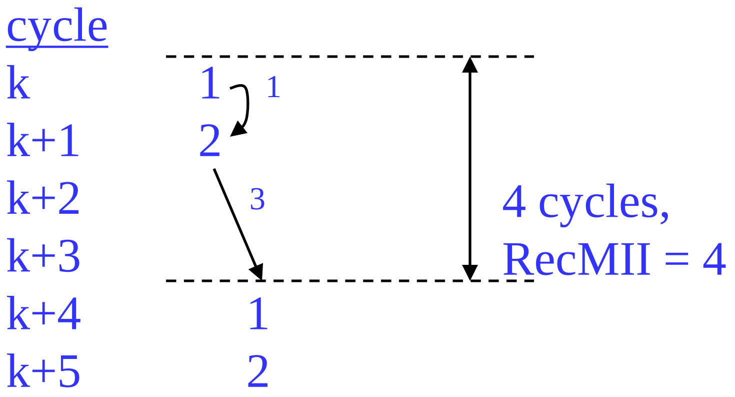
# RecMII

Approach: Enumerate all irredundant elementary circuits in the dependence graph

$$\text{RecMII} = \underset{\text{for all } c \text{ in } C}{\text{MAX}} \left(\text{delay}(c) / \text{distance}(c)\right)$$

delay(c) = total latency in dependence cycle c (sum of delays)
distance(c) = total iteration distance of cycle c (sum of distances)





cycle
k 1
k+1 2
k+2
k+3
k+4 1
k+5 2

4 cycles,
RecMII = 4

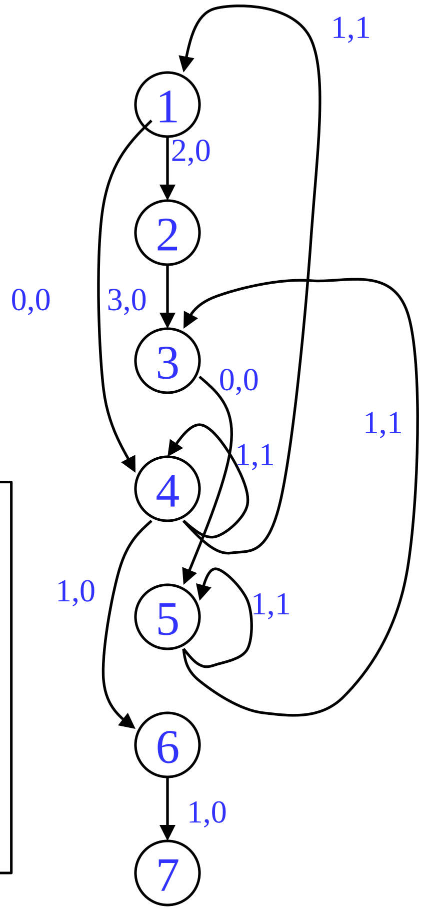
delay(c) = 1 + 3 = 4
distance(c) = 0 + 1 = 1
RecMII = 4/1 = 4

# RecMII Example

1: r3 = load(r1)
2: r4 = r3 * 26
3: store (r2, r4)
4: r1 = r1 + 4
5: r2 = r2 + 4
6: p1 = cmpp (r1 < r9)
7: brct p1 Loop

RecMII = MAX(delay(c) / distance(c))

delay(c) = total latency in dependence cycle c (sum of delays)
distance(c) = total iteration distance of cycle c (sum of distances)



<delay, distance>

4 → 4: 1 / 1 = 1
5 → 5: 1 / 1 = 1
4 → 1 → 4: 1 / 1 = 1
5 → 3 → 5: 1 / 1 = 1

RecMII = MAX(1,1,1,1) = 1

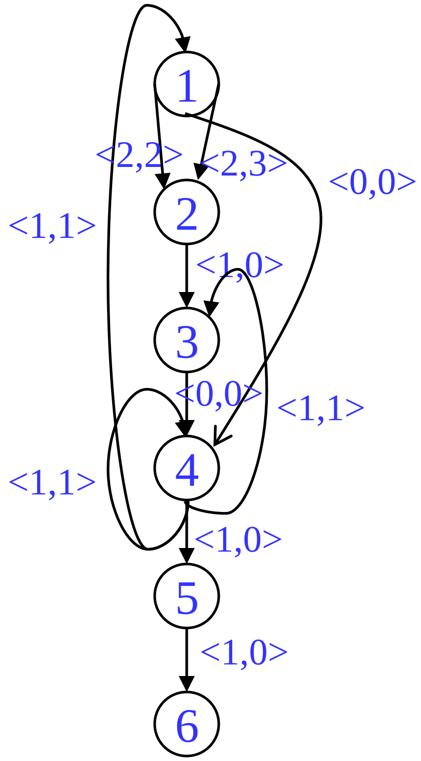
Then,

MII = MAX(ResMII, RecMII)
MII = MAX(2,1) = 2

# Homework Problem

Latencies: ld = 2, st = 1, add = 1, cmpp = 1, br = 1
Resources: 1 ALU, 1 MEM, 1 BR

```
1: r1[-1] = load(r2[0])
2: r3[-1] = r1[1] – r1[2]
3: store (r3[-1], r2[0])
4: r2[-1] = r2[0] + 4
5: p1[-1] = cmpp (r2[-1] < 100)
remap r1, r2, r3
6: brct p1[-1] Loop
```

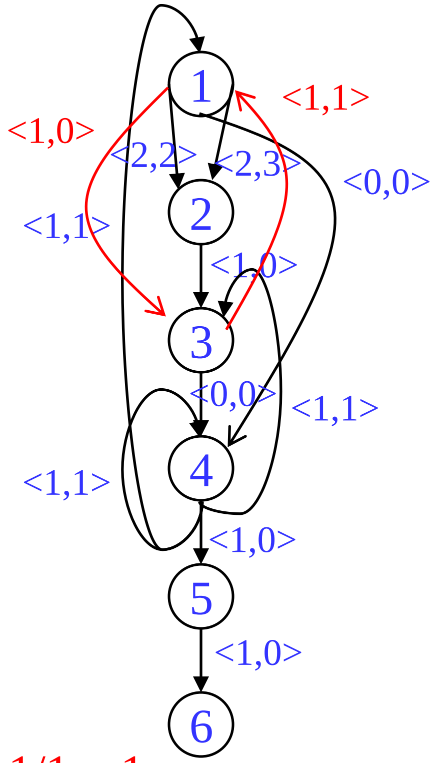


Calculate RecMII, ResMII, and MII

# Homework Problem – Answer

Latencies: ld = 2, st = 1, add = 1, cmpp = 1, br = 1
Resources: 1 ALU, 1 MEM, 1 BR

```
1: r1[-1] = load(r2[0])
2: r3[-1] = r1[1] – r1[2]
3: store (r3[-1], r2[0])
4: r2[-1] = r2[0] + 4
5: p1[-1] = cmpp (r2[-1] < 100)
remap r1, r2, r3
6: brct p1[-1] Loop
```



Calculate RecMII, ResMII, and MII

ResMII: ALU: 3 instrs / 1 unit = 3
MEM: 2 instrs / 1 unit = 2
BR: 1 instr / 1 unit = 1
MAX(3,2,1) = 3

RecMII: 4 → 4: 1/1 = 1
3 → 4 → 3: (0 + 1) / (0 +1) = 1
1 → 3 → 1: (1 + 1) / (0 + 1) = 2
1 → 2 → 3 → 1: (2+1+1) / (2+0+1) = 2
1 → 2 → 3 → 1: (2+1+1) / (3+0+1) = 1
MAX(1,1,2,2,1) = 2

MII = MAX(ResMII, RecMII) = MAX(3,2) = 3