Code Optimization: Software pipelining

ARCHI 1

Daniela Genius
daniela.genius@lip6.fr

Loop unrolling

Software pipelining

Loop unrolling

```
Loop:
     lw  R4, 0(R5)        #i1
     sll R7, R4, 1        #i2
     sw  R7, 0(R5)        #i3
     addiu R5, R5, 4      #i4
     bne R5, R9, Loop     #i5
     nop                  #i6
```

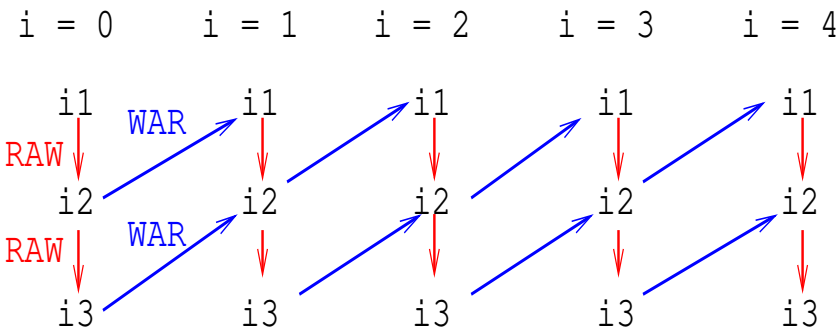
loop body

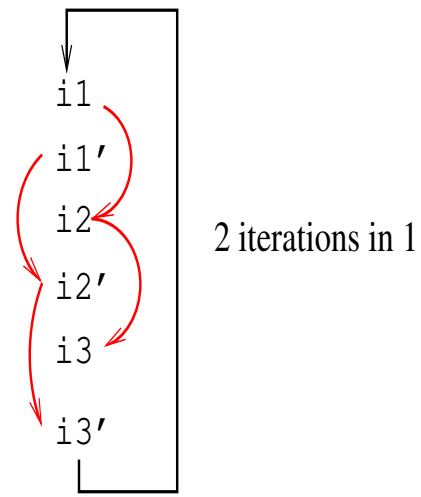
loop management

Unrolling: register renaming

to limit dependencies WAR between two (or more) iterations

to interlace instructions of different iterations

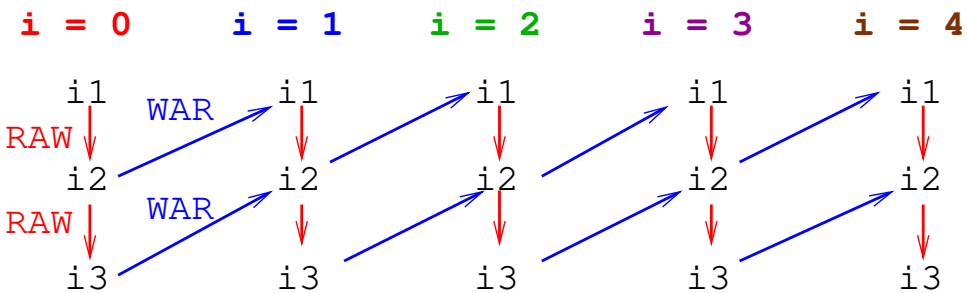




2 iterations in 1

Introduction of Software pipelining

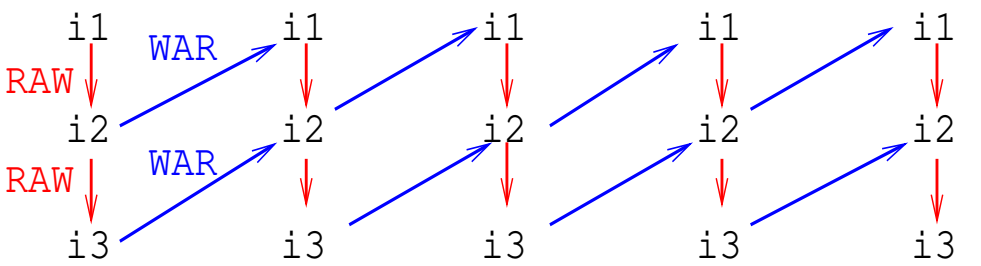
```
Loop:
      lw  R4, 0(R5)          #i1  |  loop body
      sll R7, R4, 1          #i2  |
      sw  R7, 0(R5)          #i3  |
      addiu R5, R5, 4        #i4  |  loop management
      bne R5, R9, Loop       #i5  |
      nop                    #i6  |
```



- Different viewpoint : without renaming, one can reorder instructions from several iterations
- Example for all iterations

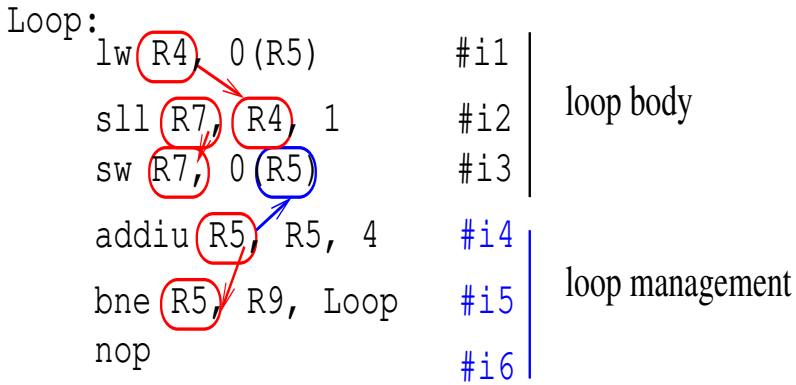
Introduction of Software pipelining

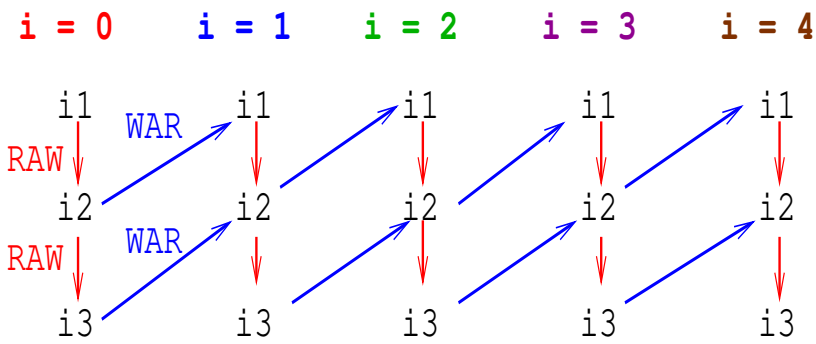
```
Loop:
      lw  R4, 0(R5)          #i1   |
      sll R7, R4, 1          #i2   | loop body
      sw  R7, 0(R5)          #i3   |
      addiu R5, R5, 4        #i4   |
      bne R5, R9, Loop       #i5   | loop management
      nop                    #i6   |
```



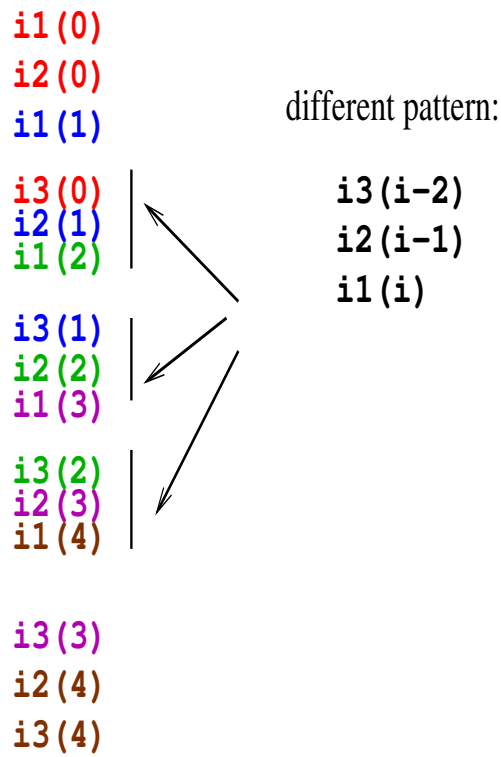

```
i1(0)
i2(0)
i1(1)

i3(0)
i2(1)
i1(2)

i3(1)
i2(2)
i1(3)

i3(2)
i2(3)
i1(4)


i3(3)
i2(4)
i3(4)
```

Introduction of Software pipelining

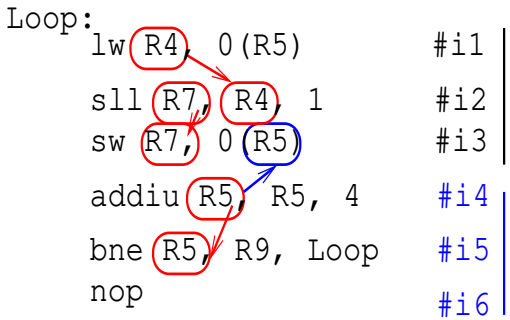


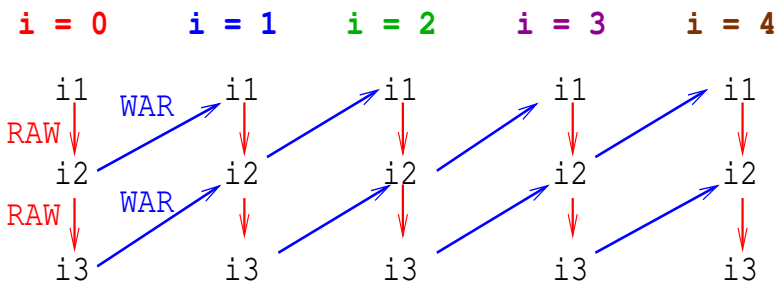


other view: without renaming one can reorder instructions from different iterations



Introduction of Software pipelining

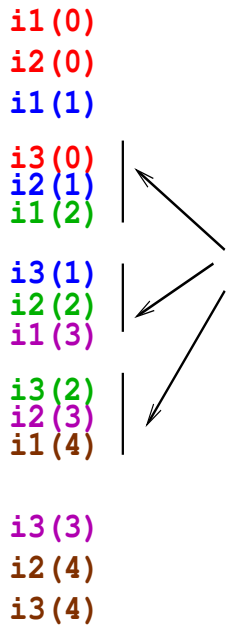




Principle of software pipelining:

build a loop with instructions stemming from different iterations

other view: without renaming one can reorder instructions from different iterations



same pattern:

```
i3(i-2)
i2(i-1)
i1(i)
```

rebuild a loop:

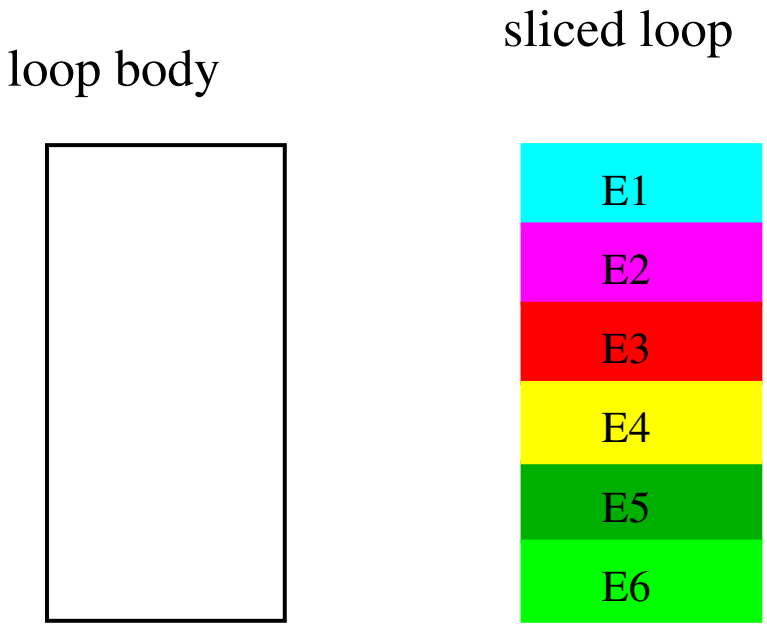
```
i3(i-2)
i2(i-1)
i1(i)
```

idea: start iteration before previous iteration has finished

Software pipelining

Same idea as instruction pipeline

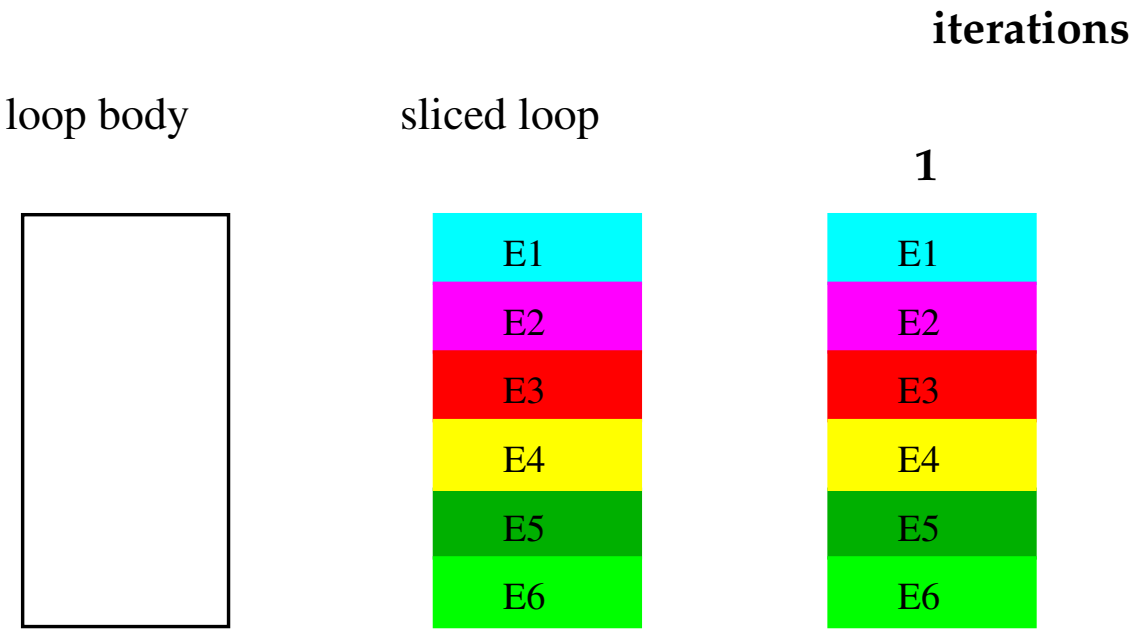
- An iteration can be sliced into `N` stages
- Stage `M` of iteration `i` can be started when iteration `i-1` has finished it. BUT program not in // thus stage `M` of iteration `i` possible when iteration `i-1` has freed ressources necessary for stage `M` (registers used) i.e. terminated the subsequent stage (`M+1`)



Software pipelining

Same idea as instruction pipeline

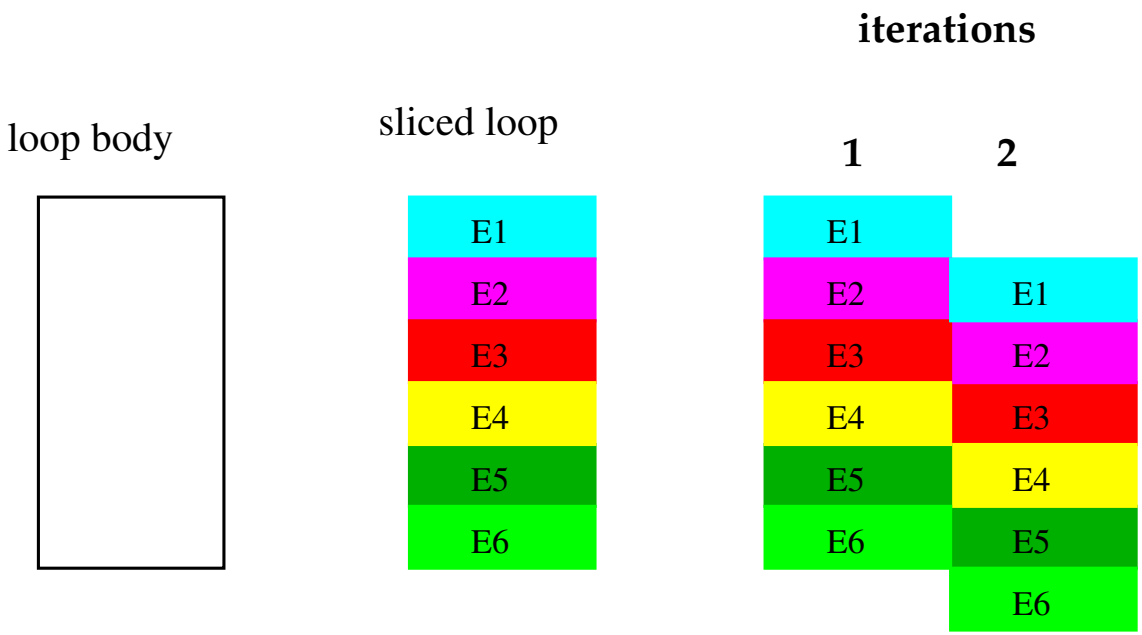
- An iteration can be sliced into `N` stages
- Stage `M` of iteration `i` can be started when iteration `i-1` has finished it. BUT program not in // thus stage `M` of iteration `i` possible when iteration `i-1` has freed ressources necessary for stage `M` (registers used) i.e. terminated the subsequent stage (`M+1`)



Le pipeline logiciel

Same idea as instruction pipeline

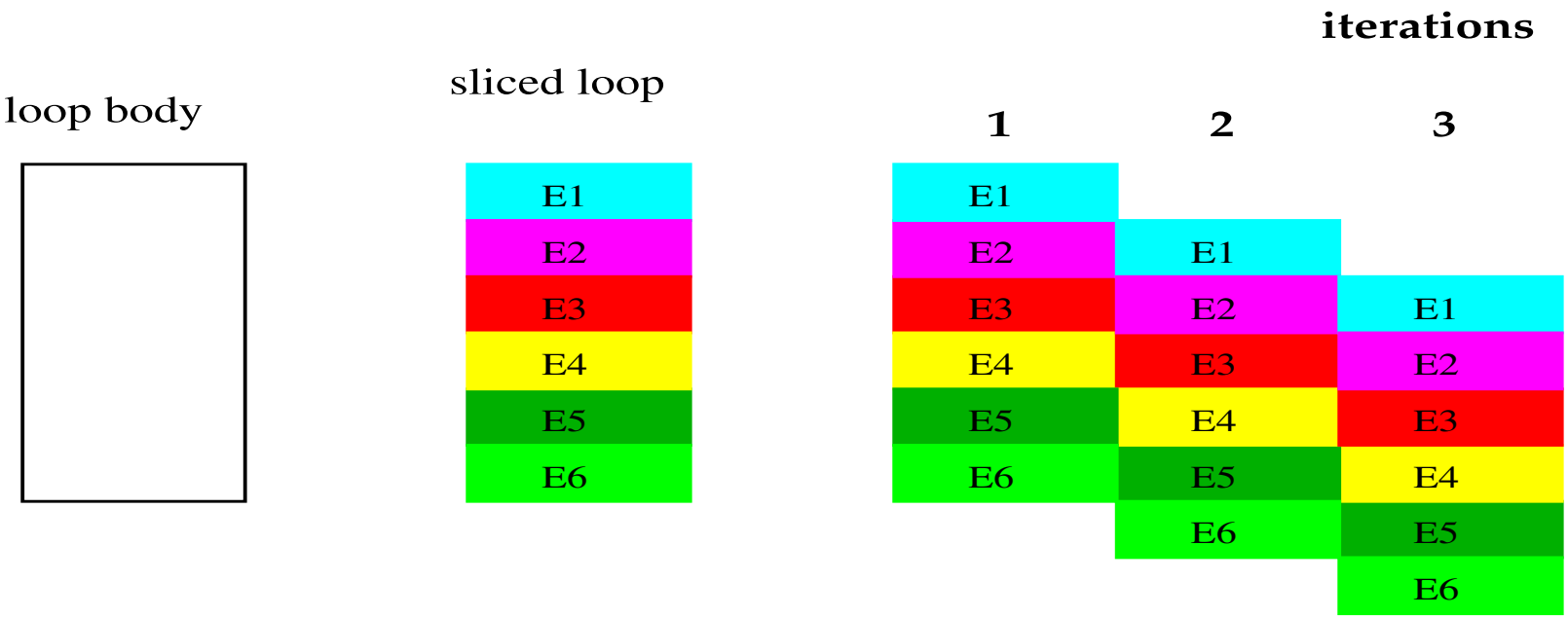
- An iteration can be sliced into `N` stages
- Stage `M` of iteration `i` can be started when iteration `i-1` has finished it. BUT program not in // thus stage `M` of iteration `i` possible when iteration `i-1` has freed ressources necessary for stage `M` (registers used) i.e. terminated the subsequent stage (`M+1`)



Software pipelining

Same idea as instruction pipeline

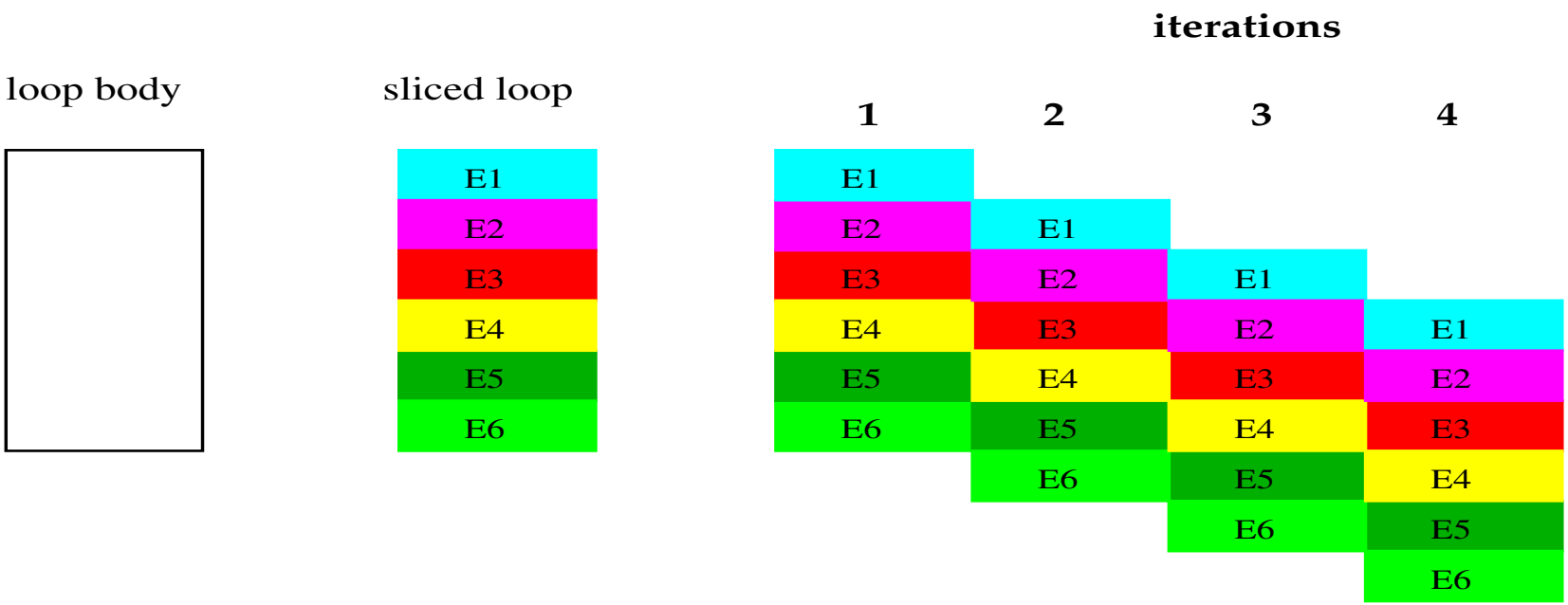
- An iteration can be sliced into `N` stages
- Stage `M` of iteration `i` can be started when iteration `i-1` has finished it. BUT program not in // thus stage `M` of iteration `i` possible when iteration `i-1` has freed ressources necessary for stage `M` (registers used) i.e. terminated the subsequent stage (`M+1`)



Software pipelining

Same idea as instruction pipeline

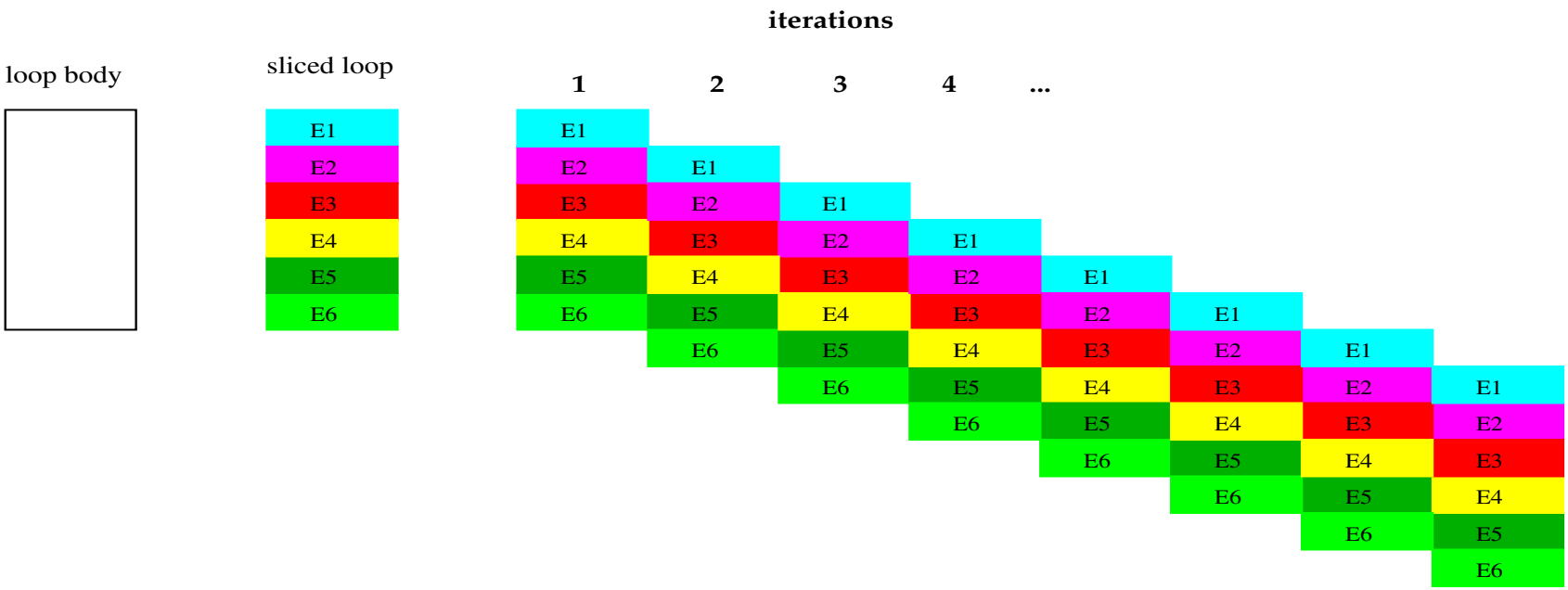
- An iteration can be sliced into `N` stages
- Stage `M` of iteration `i` can be started when iteration `i-1` has finished it. BUT program not in // thus stage `M` of iteration `i` possible when iteration `i-1` has freed ressources necessary for stage `M` (registers used) i.e. terminated the subsequent stage (`M+1`)



Software pipelining

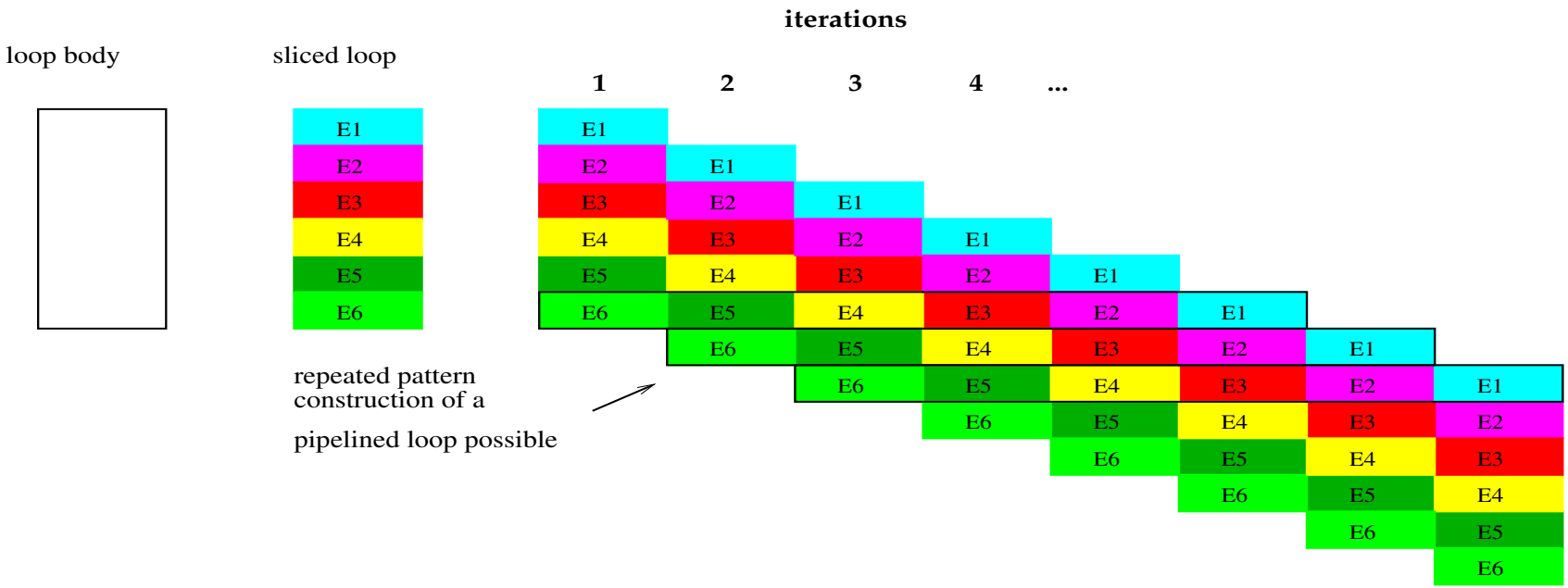
Same idea as instruction pipeline

- An iteration can be sliced into `N` stages
- Stage `M` of iteration `i` can be started when iteration `i-1` has finished it. BUT program not in // thus stage `M` of iteration `i` possible when iteration `i-1` has freed ressources necessary for stage `M` (registers used) i.e. terminated the subsequent stage (`M+1`)



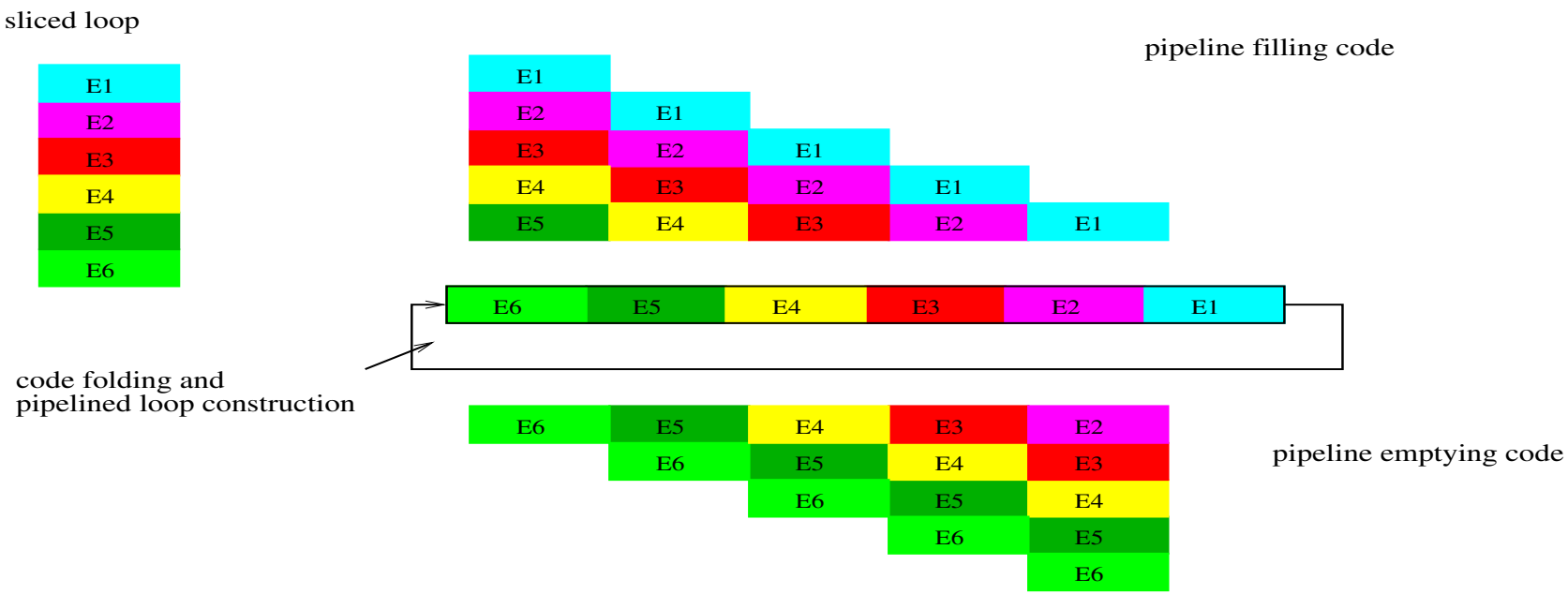
Software pipelining

- Pattern that repeats itself, different stag on N subsequent iterations
- Construction of a pipelined loop becomes possible
- Part of the initial iteration is executed in the same iteration (cf. in parallel for hardware pipeline)



Software pipelining

- Loop scheduling a stage from `N` successive itérations
- Necessity to add code for filling/emptying the pipeline



Software pipelining : example

```
 # a[i] in R6
 # a[N] in R10

loop:
   lw R4, 0(R6) # read elem i
   sll R5, R4, 1 # multiply by 2
   sw R5, 0(R6) # store elem i
   addiu R6, R6, 4
   bne R10, R6, loop
```

Preliminary step : analyze loop to determine dependencies, cycles lost or performance limitation (depending on target architecture)

Step 1 : slicing an iteration into stages

- Consider loop body and slice into stages
- Slice where there are stall cycles, dependencies, performance limitations (depending on target architecture)

```
 # a[i] dans R6
 # a[N] dans R10

loop:
   lw R4, 0(R6) # read elem i
   sll R5, R4, 1 # multiply by 2
   sw R5, 0(R6) # store elem i
   addiu R6, R6, 4
   bne R10, R6, loop
```

Step 1 : slicing an iteration into stages

- Consider loop body and slice into stages
- Slice where there are stall cycles, dependencies, Couper là où il y a des cycles de gel, des dépendances, performance limitations (depending on target architecture)
- Dependencies lw $\rightarrow$ sll $\rightarrow$ sw

```
E1 : lw R4, 0(R6)

E2 : sll R5, R4, 1

E3 : sw R5, 0(R6)
```

Step 2 : Build the new loop body

- Nouveau corps = 1 étape pour n itérations successives si découpage en n étapes
- Respecter l'ordre des étapes des différentes itérations : itérations les plus anciennes en 1er
- Si 1ère étape E_1 correspond à itération i : $E_n(i-n+1)$, $E_{n-1}(i-n+2)$,..., $E_2(i-1)$, $E_1(i)$
- `R6=a[i]` incremented in each iteration, adjust instructions using it, if required

```
E3(i-2) : sw R5, ??(R6)

E2(i-1) : sll R5, R4, 1

E1(i) : lw R4, ??(R6)
```

Step 2 : Build the new loop body

- `R6` incremented by 4 in each iteration, must adjust instructions using it
- `R6` contains address of element i

```
E3(i-2) : sw R5, -8(R6) # elem i-2 is located -2x4
                        # bytes in memory before elem i
E2(i-1) : sll R5, R4, 1

E1(i) : lw R4, 0(R6) # elem i, address in R6
```

Step 3 : Insert loop management

- Stop at adequate moment
- Stop when last element has been read (stage 1)
- Stp at address of element N ant start at i=2
- R6 must contain addresse of second element before the loop

```
loop: # de 2 a N-1
   sw R5, -8(R6)  # E3(i-2)
   sll R5, R4, 1  # E2(i-1)
   lw R4, 0(R6)   # E1(i)
   addiu R6, R6, 4
   bne R6, R10, loop # R10 contains @tab[N]
   nop
```

Step 3 : insert prologue and epilogue

- Stop at adequate moment
- Stop when last element has been read (stage 1)
- Stp at address of element N ant start at i=2
- R6 must contain addresse of second element before the loop

```
# before : E1(0), E2(0), E1(1)
#       R6 = @tab[2]
loop: # de 2 a N-1
   sw R5, -8(R6)  # E3(i-2)
   sll R5, R4, 1  # E2(i-1)
   lw R4, 0(R6)   # E1(i)
   addiu R6, R6, 4
   bne R6, R10, loop # R10 contains @tab[N]
   nop
# after : E3(N-2), E2(N-1), E3(N-1)
```

Reordering (1)

- Reorder instructions to avoid nop and stall cycles
- No more intra-iteration RAW dependencies between instructions in loop body
- Inter-iteration RAW dependencies

```
# before : E1(0), E2(0), E1(1)
#       R6 = @tab[2]'
loop : # i de 2 a N-1
  sw R5, -8(R6) # E3(i-2)
  sll R5, R4, 1 # E2(i-1)
  lw R4, 0(R6)  # E1(i)
  addiu R6, R6, 4
  bne R6, R10, loop # R10 contains @tab[N]
  nop
# after : E3(N-2), E2(N-1), E3(N-1)
```

Reordering (2)

- Fill the delayed slot : `lw` in delayed slot
- Eliminate stall cycle `addiu` → `bne` : raise the `addiu`
- 1 iteration = 5 cycles on MIPS32

```
# before : E1(0), E2(0), E1(1)
#          R6 = @tab[2]
loop :
  # i de 2 a N-1
  sw R5, -8(R6)       # E3(i-2)
  addiu R6, R6, 4
  sll R5, R4, 1       # E2(i-1)
  bne R6, R10, loop # R10 contains @tab[N]
  lw R4, -4(R6)       # E1(i)

# after : E3(N-2), E2(N-1), E3(N-1)
```

Software pipelining : example 2

```
# a[] in R6
# a[N] in R10

loop:
  lw R4, 0(R6) # read val elem i
  sll R5, R4, 1 # val * 2
  add R5, R5, R4 # val * 2 + val == val * 3
  sw R5, 0(R6) # store elem i
  addiu R6, R6, 4
  bne R10, R6, loop
  nop
```

- Analysis of loop body + slicing

Software pipelining : example 2

```
 # a[] in R6 '
 # a[N] in R10 '

loop:
   lw R4, 0(R6) # read val elem i
   sll R5, R4, 1 # val * 2
   add R5, R5, R4 # val * 2 + val == val * 3
   sw R5, 0(R6) # store elem i
   addiu R6, R6, 4
   bne R10, R6, loop
   nop
```

- Analysis of loop body + slicing
- RAW dependencies : lw $\rightarrow$ sll $\rightarrow$ add $\rightarrow$ sw
- Slice into 4 stages

Software pipelining : example 2

- RAW dependencies : lw $\rightarrow$ sll $\rightarrow$ add $\rightarrow$ sw
- Slice into 4 stages
- Pipelined loop :

```
 # a[i] in R6
 # a[N] in R10

loop:
   sw R5, -12(R6) # E4(i-3) store elem i-3
   add R5, R5, R4  # E3(i-2) elem i-2 * 3
   sll R5, R4, 1   # E2(i-1) elem i-1 * (2 + 1)
   lw R4, 0(R6)    # E1(i) read elem i

   addiu R6, R6, 4
   bne R10, R6, loop
   nop
```

Is this loop correct ?

Software pipelining : example 2

```
# a[] in R6
# a[N] in R10
loop:
   sw R5, -12(R6) # E4 : store elem i-3
   add R5, R5, R4  # E3 : elem i-1 * (2 + 1)
   sll R5, R4, 1   # E2 : elem i-2 * 2
   lw R4, 0(R6)    # E1 : read elem i

   addiu R6, R6, 4
   bne R10, R6, loop
```

- Is this loop correct ?
- E1 produces R4 used in E2 and E3
- E2 produces R5 used in E3
- E3 uses R5 and R4 and produces R5 for E4
- Explained on blackboard...

Software pipelining : example 2

```
# a[] in R6
# a[N] in R10
loop:
   sw R5, -12(R6) # E4 : store elem i-3
   add R5, R5, R4  # E3 : elem i-1 * (2 + 1)
   sll R5, R4, 1   # E2 : elem i-2 * 2
   lw R4, 0(R6)    # E1 : read elem i

   addiu R6, R6, 4
   bne R10, R6, loop
```

- Is this loop correct ?
- E1 produces R4 used in E2 and E3
- E2 produces R5 used in E3
- E3 uses R5 and R4 and produces R5 for E4
- Explained on blackboard...

Software pipelining : example 2

```
# a[] in R6
# a[N] in R10
loop:
   sw R5, -12(R6) # E4 : store elem i-3
   add R5, R5, R4  # E3 : elem i-1 * (2 + 1)
   sll R5, R4, 1   # E2 : elem i-2 * 2
   lw R4, 0(R6)    # E1 : read elem i

   addiu R6, R6, 4
   bne R10, R6, loop
```

- Is this loop correct ?
- E1 produces R4 used in E2 and E3
- E2 produces R5 used in E3
- E3 uses R5 and R4 and produces R5 for E4
- Explained on blackboard...

Software pipelining : example 2

```
# a[] in R6
# a[N] in R10
loop:
   sw R5, -12(R6) # E4 : store elem i-3
   add R5, R5, R4  # E3 : elem i-1 * (2 + 1)
   sll R5, R4, 1   # E2 : elem i-2 * 2
   lw R4, 0(R6)    # E1 : read elem i

   addiu R6, R6, 4
   bne R10, R6, loop
```

- Is this loop correct ?
- E1 produces R4 used in E2 and E3
- E2 produces R5 used in E3
- E3 uses R5 and R4 and produces R5 for E4
- Explained on blackboard...

Software pipelining : example 2

```
# a[] in R6
# a[N] in R10
loop:
   sw R5, -12(R6) # E4 : store elem i-3
   add R5, R5, R4  # E3 : elem i-1 * (2 + 1)
   sll R5, R4, 1   # E2 : elem i-2 * 2
   lw R4, 0(R6)    # E1 : read elem i

   addiu R6, R6, 4
   bne R10, R6, loop
```

- Is this loop correct ?
- E1 produces R4 used in E2 and E3
- E2 produces R5 used in E3
- E3 uses R5 and R4 and produces R5 for E4
- Explained on blackboard...

Software pipelining : back to slicing

- Each stage must write a value to a register different from the others : if this is not the case, rename
- Each stage must use the values of the preceding stage and produce only for the subsequent stage : otherwise, insert instructions to pass the values via different registers at each intermediate stage (or revise the slicing into stages)

```
 # a[i] in R6, a[N] in R10
loop:
   sw R7, -12(R6)       # E4(i-3) store elem i-3
   add R7, R5, R9       # E3(i-2) elem i-1 * (2 + 1)
   ori R9, R4, 0        # E2(i-1) save R4
   sll R5, R4, 1        # E2(i-1) elem i-2 * 2
   lw R4, 0(R6)         # E1(i) read elem i
   addiu R6, R6, 4
   bne R10, R6, loop
```

Software pipelining : what about internal control flow ?

Example 3

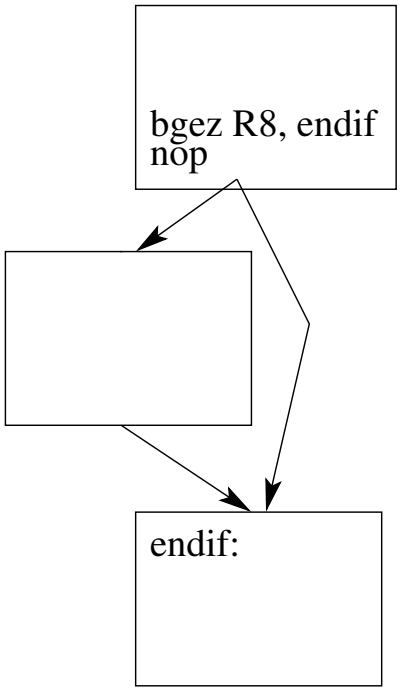
```
# R5 = @tab[i], R7 = @tab[N]
loop:
    lw   R8, 0(R5)
    bgez R8, endif
    nop
    sub  R8, R0, R8
    sw   R8, 0(R5)
endif:
    addiu R5, R5, 4
    bne  R7, R5, loop
    nop
```

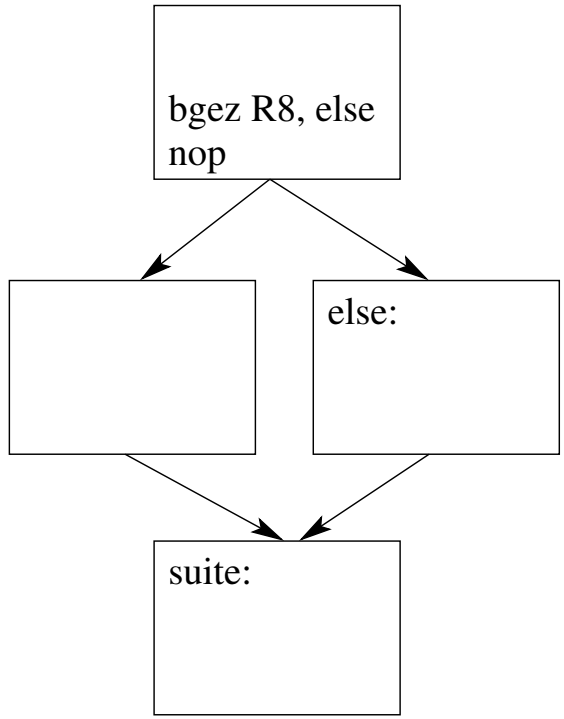
Example 4

```
# R5 = @tab[i], R7 = @tab[N]
loop:
    lw    R8, 0(R5)
    bgez  R8, else
    nop
    addi  R8, R8, -1
    j suite
    nop
else:
    addi R8, R8, 1
suite:
    sw    R8, 0(R5)
    addiu R5, R5, 4
    bne   R7, R5, loop
    nop
```

Software pipelining : what about internal control flow ?

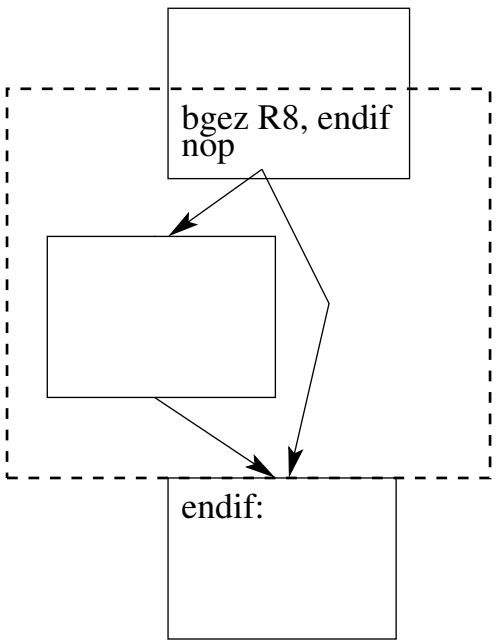
- Where to slice the iteration ?

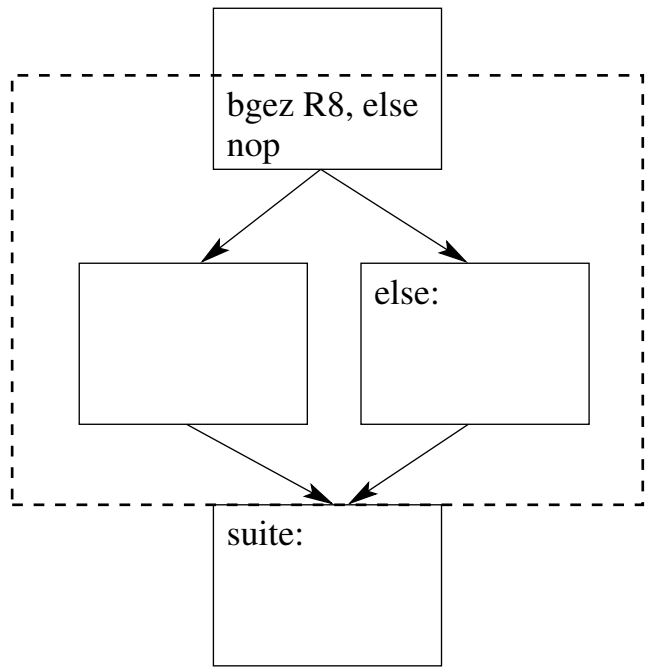




Software pipelining : what about internal control flow ?

- Cannot separate a branch instruction from the instructions whose execution depends on it, cannot tell the subsequent iteration whether to execute them or not





Summary : Where to slice the iteration ?

- Consider only the loop body
- Instructions depending on a branch (control dependency) must be in the same stage as the branch statement
  Only dominant blocks are sliced
- Slice where there is performance loss, dependency of type RAW or stall cycle (depends on architecture)
- Once slicing realized, verify :
  - Each stage provides values exclusively to the subsequent stage, otherwise add instructions to pass the values or revise the slicing into stages.
  - Each stage writes to registers different from the other stages, otherwise loss/incoherency of values.