Computer Architecture

R. Poss
Computer Systems Architecture group (UvA)
e-mail: r.c.poss@uva.nl

Pipeline reminders

Motivation

- Pipeline-level parallelism is the weapon of architects
  to **increase throughputs**
  and **tolerate latencies of communication**
  for **individual instruction streams**
  (i.e. sequential programs)
  without participation from the programmer
  (i.e. implicit)
- We will cover true and explicit parallelism later in the course

Processor performance

- Latency: expressed as **CPI** = cycles per instruction
  divide by frequency to obtain absolute latency
- Throughput: expressed as **IPC** = instructions per cycle
  multiply by frequency to obtain absolute throughput
- Pipelining objective: **increase IPC**, also decrease CPI
  As we will see ↘CPI and ↗IPC are conflicting requirements

Types of pipeline concurrency

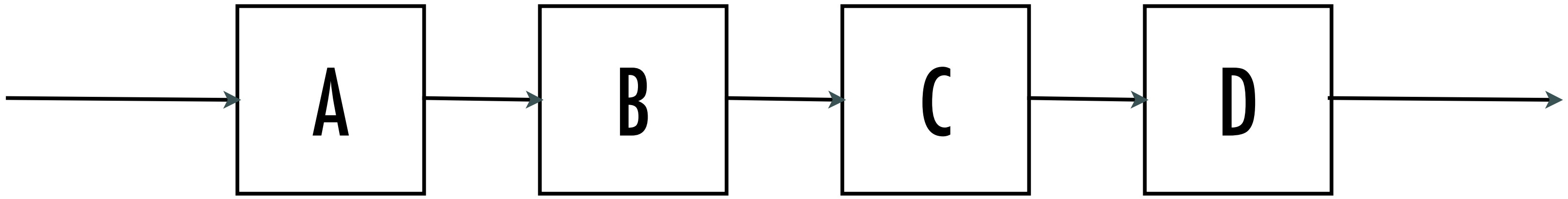
- **Pipelined**: operations broken down in sub-tasks
  ⇒ different sub-tasks from different operations run in parallel

- **Scalar** pipelined: multiply the functional units
  ⇒ the same sub-task from different operations run in parallel

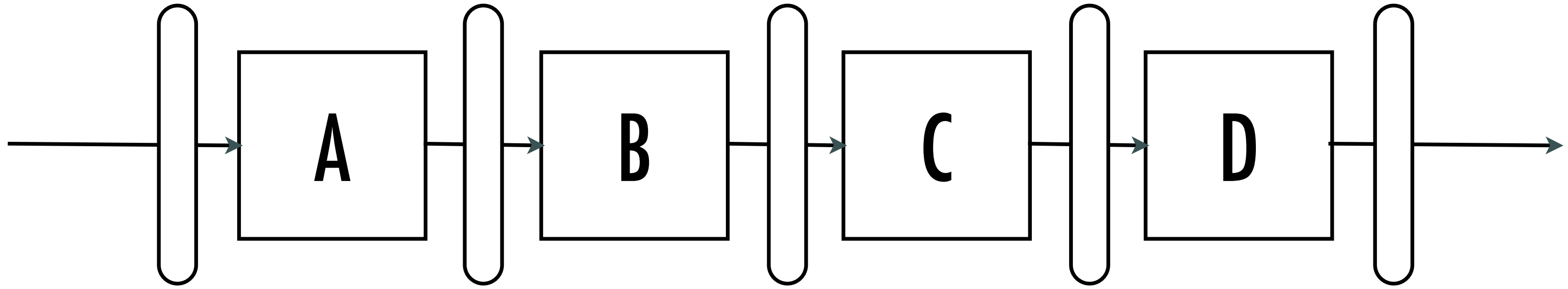
- **Superscalar** pipelined: multiply the issue units
  ⇒ multiple operations issued and completing simultaneously

Pipelines

- Each instruction / operation can be decomposed in sub-tasks:
  Op = A ; B ; C ; D

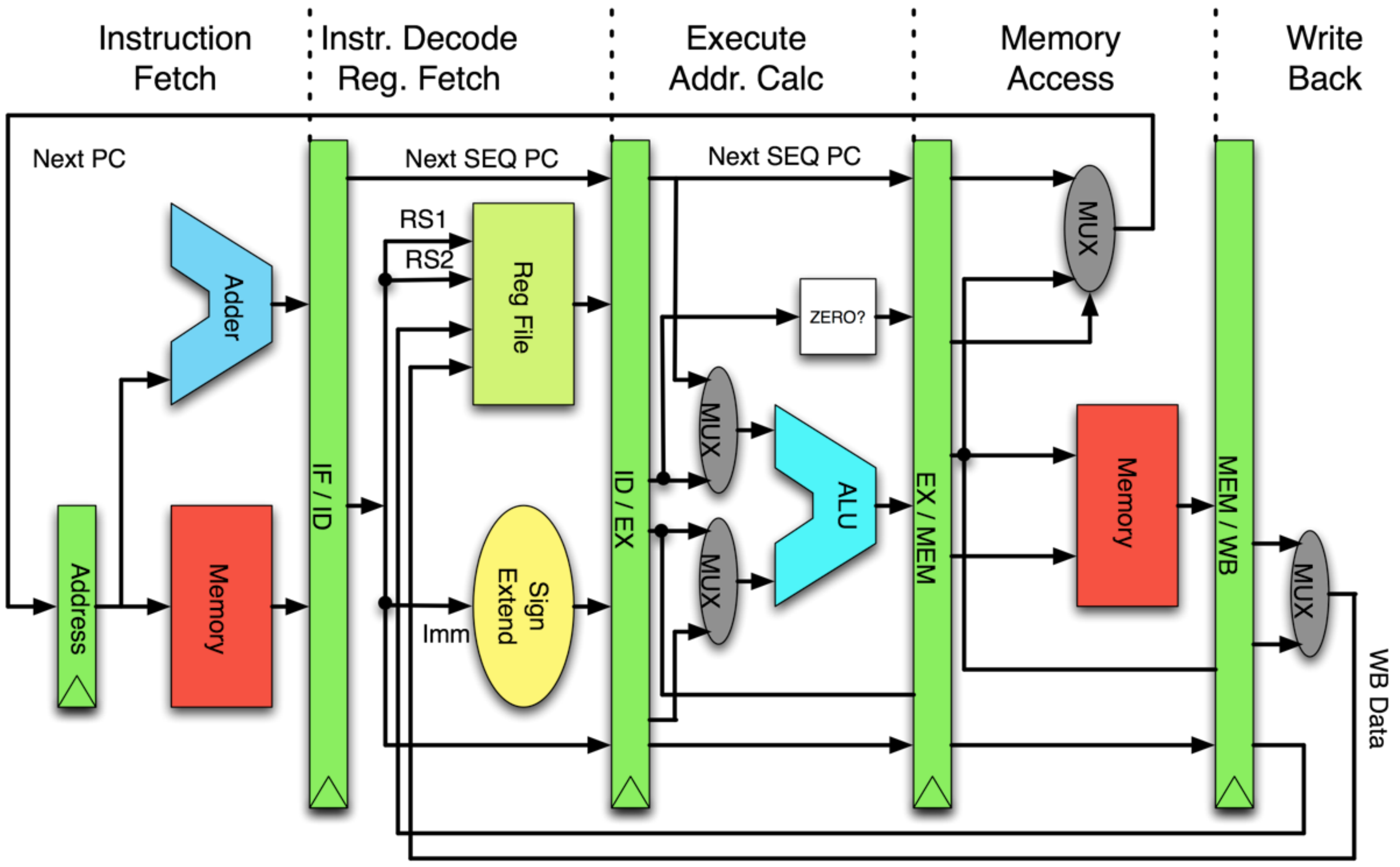


- Considering an instruction stream [Op_1; Op_2; ...]
  at each cycle n we can run in parallel: A_{n+3} || B_{n+2} || C_{n+1} || D_n



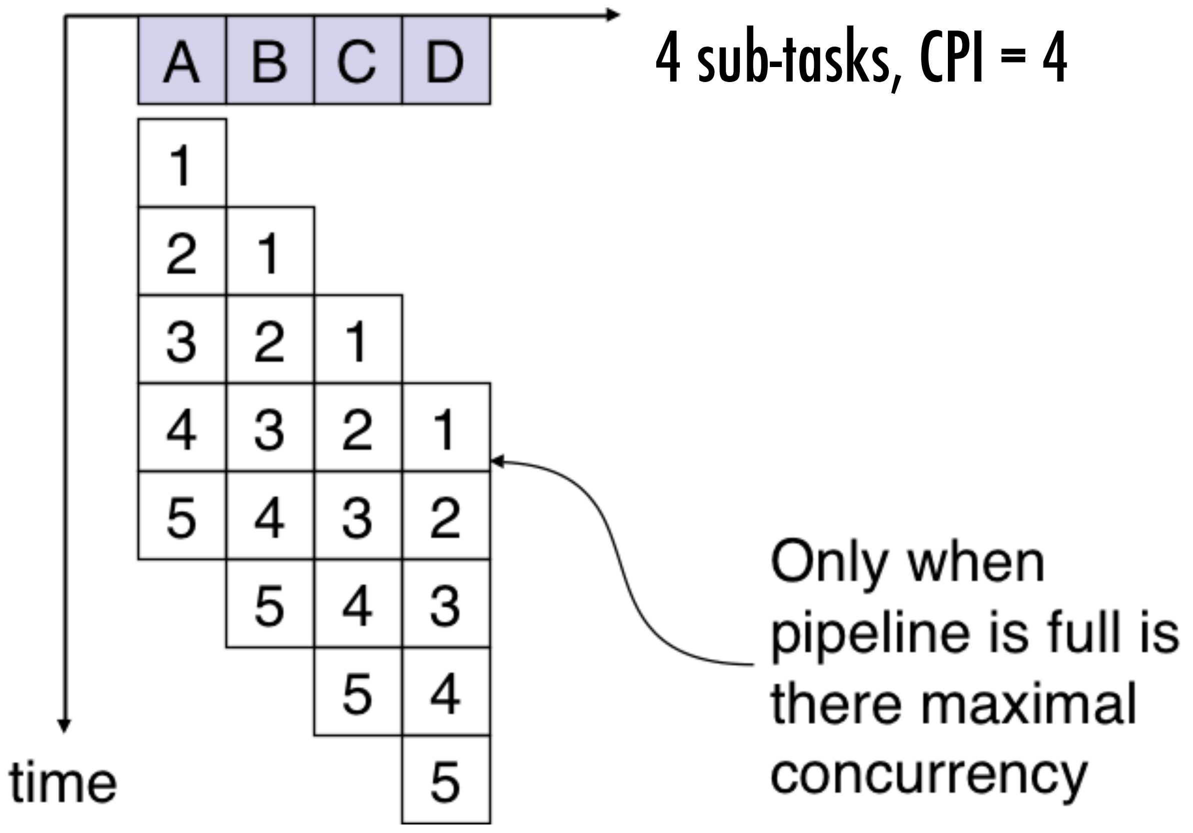
at start of cycle n: input A_{n+3} input B_{n+2} input C_{n+1}

Example: MIPS



Dynamic behavior

- 1 program, 5 operations:



4 sub-tasks, CPI = 4

(total execution time: 8 cycles)

Pipeline performance

- This pipeline has a length of 4 subtasks,
  assume each sub-task takes t seconds
  - for a single operation we get no speedup; it takes 4t seconds to complete all of the subtasks
  - this is the same as performing each sub task in sequence on the same hardware
- In the general case – for n operations – it takes 4t seconds to produce the first result and t seconds for each subsequent result

Pipeline performance

- For a pipeline of length L and cycle time t, the time T it takes to process n operations is:
  $T(n) = L \cdot t + (n-1) \cdot t = (L-1) \cdot t + n \cdot t$
- We can characterise all pipelines by two parameters:
  - **startup time**: $S = (L-1) \cdot t$ (unit: seconds)
  - **maximum rate**: $r_\infty = 1/t$ (unit: instructions per second)
- Also used:
  - **half-performance vector length** $n_{½}$ which verifies $T(n_{½})/2 = n_{½} \cdot r_\infty^{-1}$
    $n_{½} = L-1 = S/t$ (unit: number of operations)

Optimization strategies

- Long instruction sequences suggest IPC = 1
- However there are problems with this:
  - some instructions require less sub-tasks than others
  - hazards: dependencies and branches
  - long-latency operations: can’t fit the pipeline model
- What to do about these? The rest of the lecture covers this.

Trade-offs

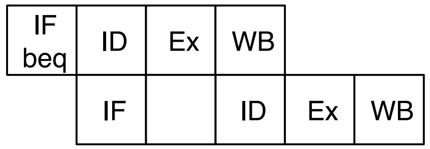
- Observations:
  - **more complexity** per sub-task requires **more time per cycle**
  - conversely, as the sub-tasks become simpler the cycle time can be reduced
  - so **to increase the clock rate** instructions must be **broken down into smaller sub-tasks**
  - ...but operations have a fixed complexity
  - **smaller sub-tasks mean deeper pipelines** = more stages
    $\Rightarrow$ more instructions need to be executed to fill the pipeline

Control hazards

Control hazards

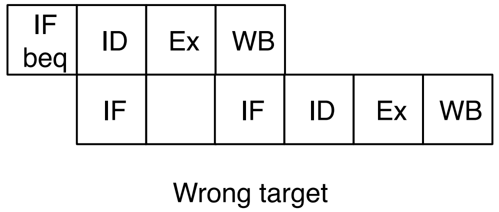
- Branches – in particular **conditional branches** – cause pipeline hazards
- the outcome of a conditional branch is not known until the end of the EX stage, but is required at IF to load another instruction and keep the pipeline full
  - A simple solution:
    assume by default that the branch falls through – i.e. is not taken – then continue speculatively until the target of the branch is known

<u>Branch not taken</u>



Continue to fetch but stall ID until branch target is known
– one cycle lost

<u>Branch is taken</u>

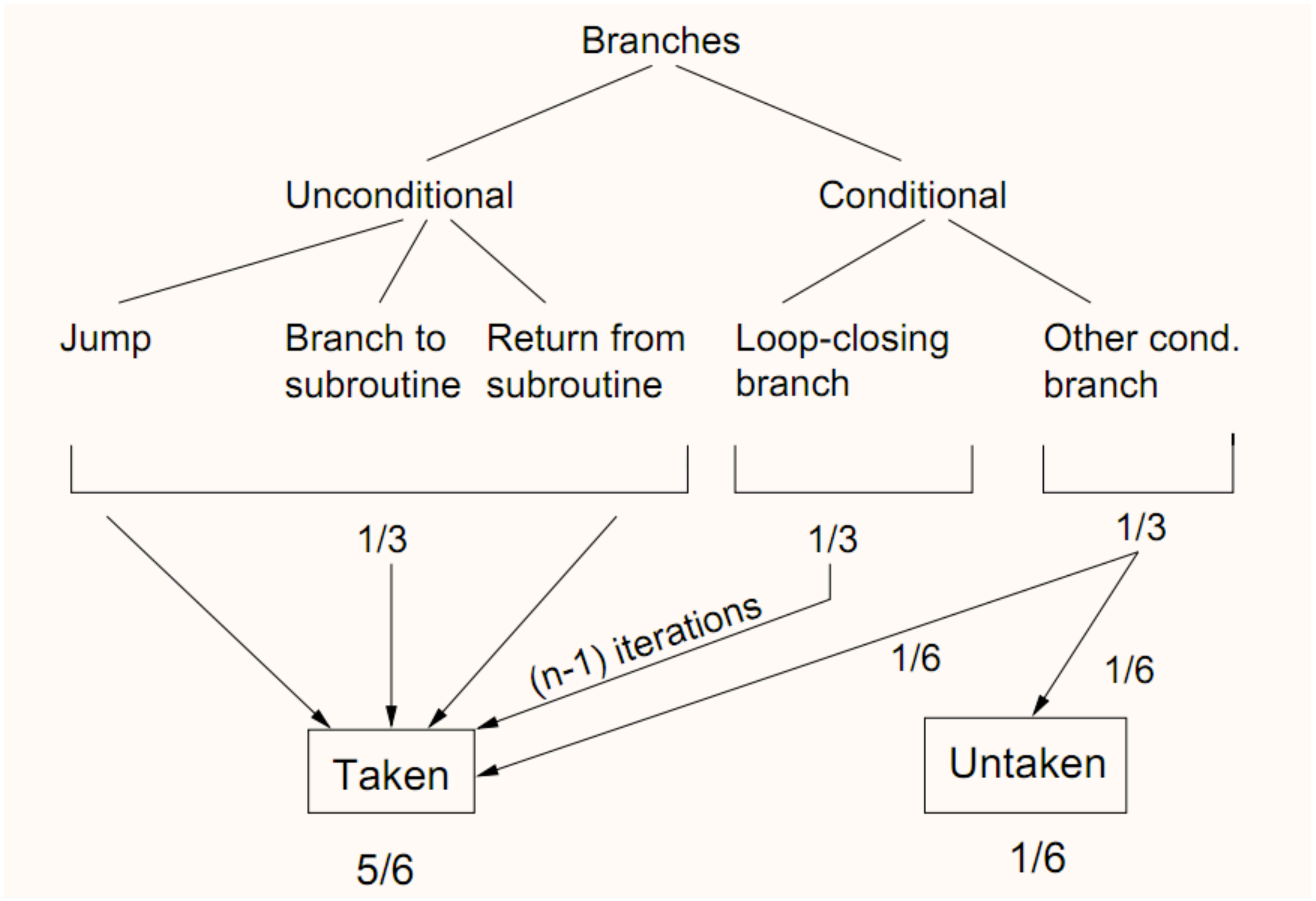


Need to refetch at new target
– two cycles lost

How to overcome

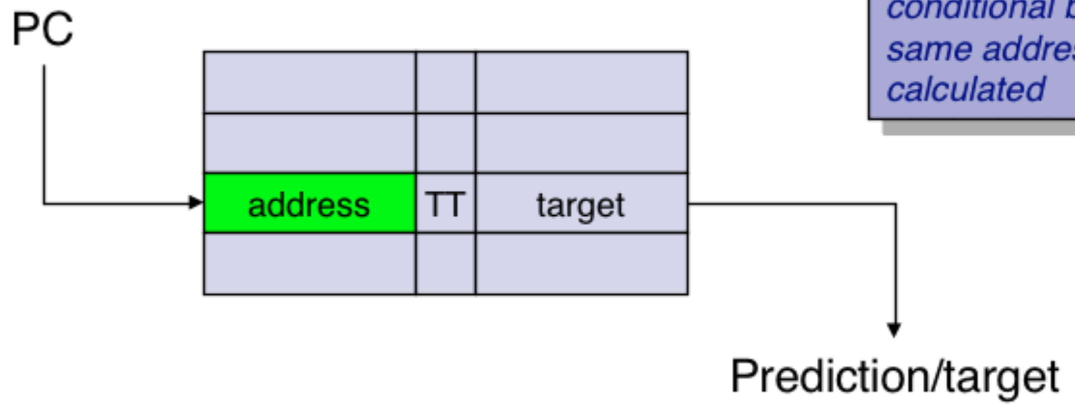
- Eliminate branches altogether via **predication** (most GPUs)
- Expose the branch delay to the programmer / compiler: **branch delay slots** (MIPS, SPARC, PA-RISC)
- Fetch from both targets, requires branch target address prediction
- Predict whether the branch is taken or not: **branch prediction**
- Execute instructions from other threads: **hardware multithreading** (eg Niagara, cf next lecture)

Grohoski's estimate



Branch prediction using history buffers

- Stores the **prediction state in a table**, either associatively addressed or indexed on small number of address bits
  - Can also store branch target if it is associative
  - Get prediction at IF stage and update prediction when condition is resolved

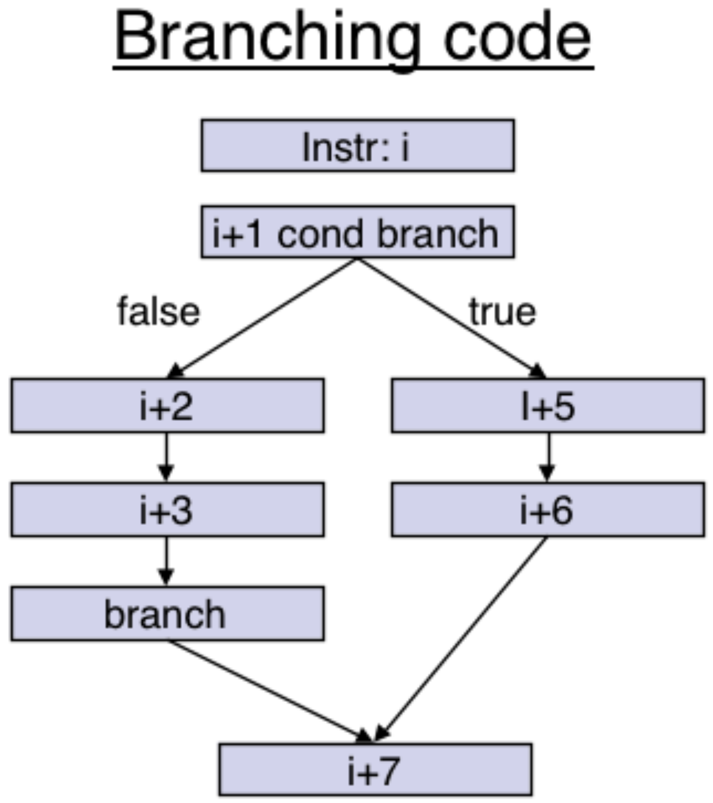


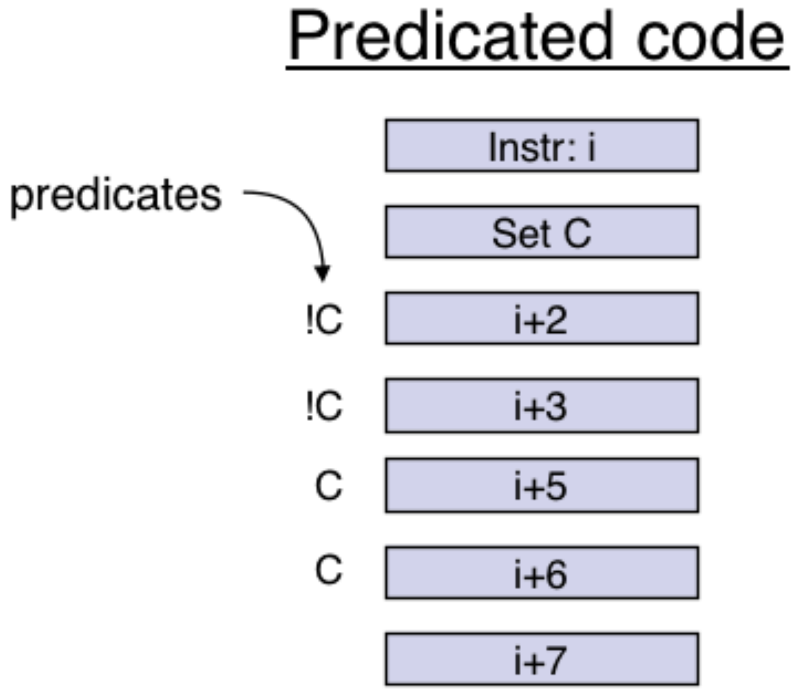
N.b. in a loop, on every conditional branch, the same address is calculated

Predication

- Control flow can (in some cases) be replaced by guarded or predicated instruction execution...
  - a condition sets a predicate register (boolean)
  - instructions are predicated on that register
  - any state change (WB or Mem write) only occurs if the predicate is true
- Useful in long pipelines where branch hazards can be costly,
  or to simplify the pipeline logic by not handling control hazards at all
- it removes a control hazard at the expense of **redundant instruction execution**

Predication – example





No hazards but redundant execution

Branch delay slots

- Specify in the ISA that a branch takes effect two instructions later, then let the compiler / programmer fill the empty slot

```
L1:
   lw a x[i]
   add a a a
   sw a x[i]
   sub i i 4
   bne i 0 L1
   nop
L2:
```

1 cycle wasted at each iteration

```
L1:
   lw a x[i]
   add a a a
   sw a x[i]
   bne i 4 L1
   sub i i 4
L2:
```

no bubble, but one extra sub at last iteration

Fetch from both targets

- Using an additional I-cache port, both taken and not-taken are fetched
  - Then at EX a choice is made as to which is decoded
- When coupled with a branch delay slot this eliminates all wasted cycles, but...
  - longer pipelines, eg 20 stages, might contain several branches in the pipe prior to EX
  - **multiple conditional branches will break** this solution,
  - as every new branch doubles the number of paths fetched

Summary

- **Control hazards** are situations where the pipeline is not fully utilized due to branch instructions
- **Branch prediction**, **delay slots** and **predication** are architectural solutions to overcome control hazards
  - Only branch prediction is fully invisible to software
  - Without these solutions, software must avoid branches by inlining and loop unrolling; with a trade-off: more code means more pressure on I-Cache