Imperial College London

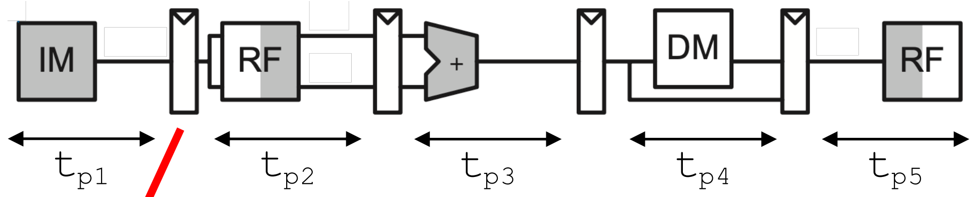
Lecture 11

Additional Topics

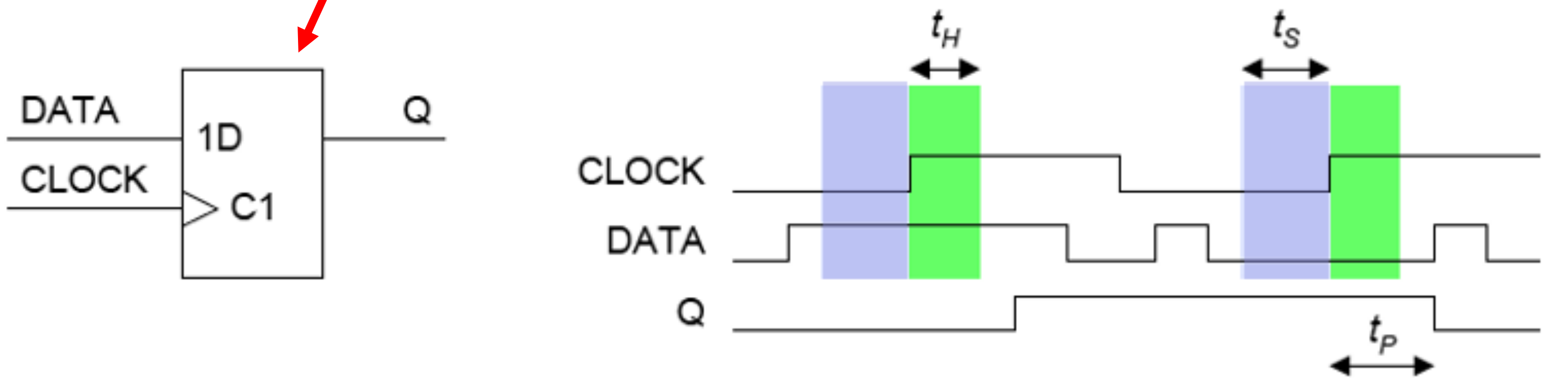
Peter Cheung
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/EE2_CAS/
E-mail: p.cheung@imperial.ac.uk

5-stage Pipelining



The DATA input to a flipflop or register must not change at the same time as the CLOCK.

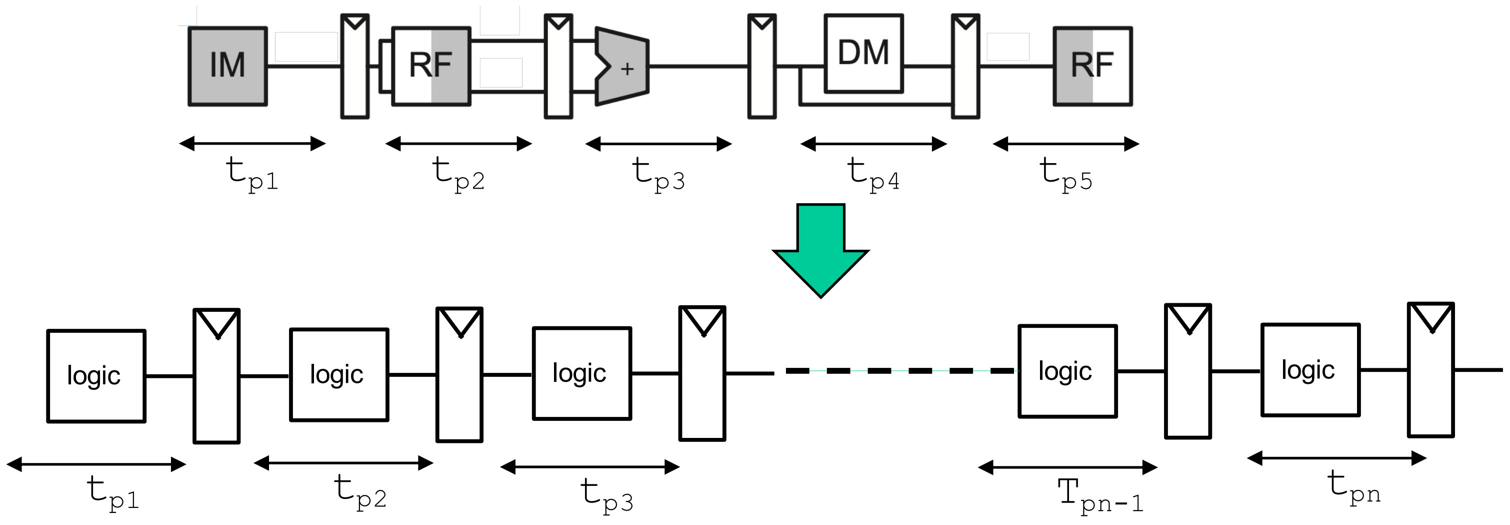


Setup Time: DATA must reach its new value at least t_S before the CLOCK↑ edge.

Hold Time: DATA must be held constant for at least t_H after the CLOCK↑ edge.

Maximum processor clock frequency: $$\frac{1}{\max(t_{p1},t_{p2},t_{p3},t_{p4},t_{p5})+t_s}$$

Deep Pipelining

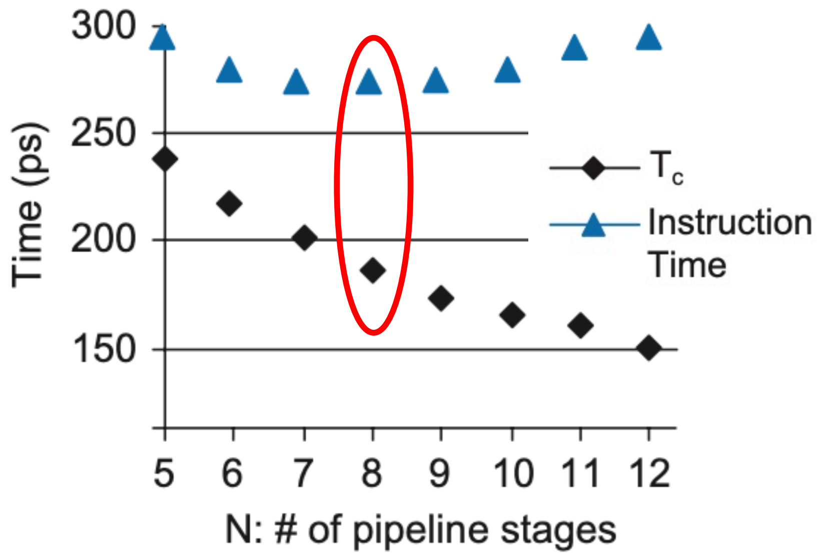


- Cycle per instruction (CPI) for pipelined processor > 1 (e.g. 1.25), but higher clock frequency.
- Increase clock frequency by adding more pipeline stages by reducing worst-case t_p.
- Deeper pipeline creates more data and control hazards, and more complex detection/mitigation hardware.
- Register setup time also results in diminishing return.
- Example: 2015 Intel i7 uses 19-stage pipeline; ARM processor typically uses 13—stage pipeline.

An Example on Pipelining

- A single-cycle processor with a propagation delay of 750ps is to be pipelined into N stages.
- Assume:
  - Register overhead (i.e. setup time) is 90ps;
  - Adding a pipeline stage does not increase hazard logic delay;
  - 5 stage pipeline would result in a CPI of 1.25;
  - Each additional pipeline stage add 0.1 to CPI due to branch and other hazards (stalling).
- How many pipeline stages gives best performance?

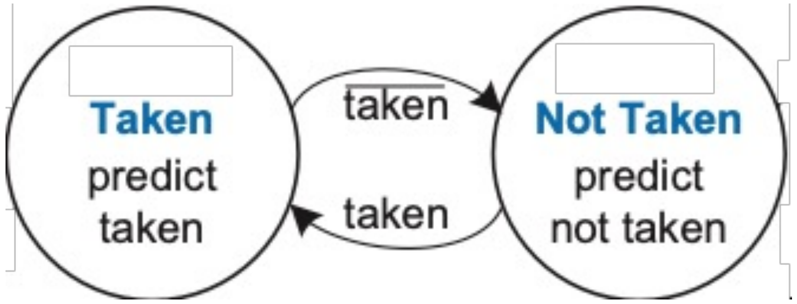
- Cycle time (i.e. clock period) is: $T_c = \frac{750}{N} + 90$ ps.
- CPI = 1.25 + 0.1(N-5), for N ≥ 5.
- Instruction time = $CPI \times T_c$



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

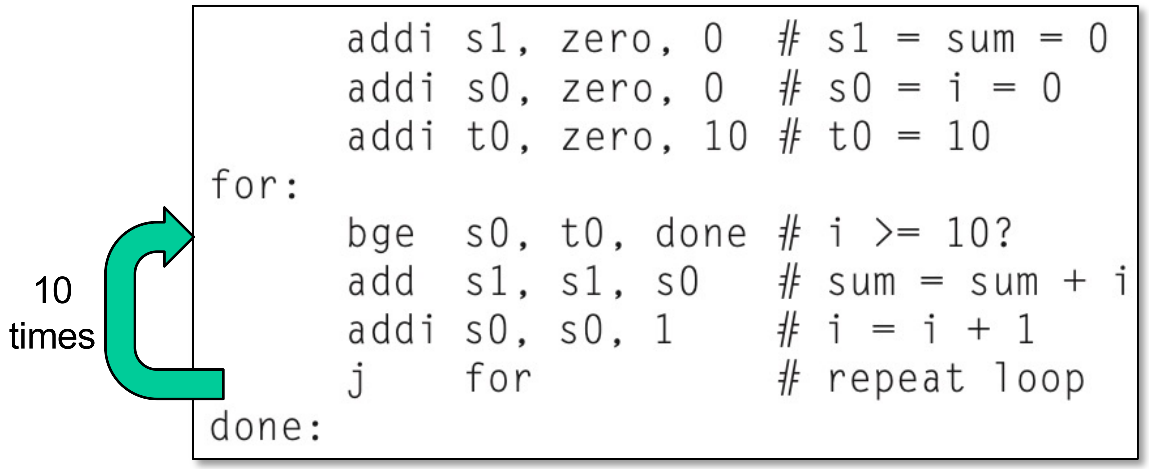
Simple branch prediction

- So far, all branch instruction are assumed **NOT TAKEN**.
- Increased pipeline stages results in higher penalty (flushing) if branch **IS TAKEN**.
- Improve performance by adding **ACCURATE** branch prediction.
- **STATIC** branch prediction – forward branch assumes **NOT TAKEN**; backward branch assumed **TAKEN**.
- **SIMPLE DYNAMIC** branch prediction – due historical information for prediction. The simples is: **Branch taken last time, predict will also be taken next time**.
- Maintain a table of branch instructions and what happened most recently.
- The table is known as a **branch target buffer** which includes destination address of branch and 1-bit history.



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Two-bit Branch Predictor



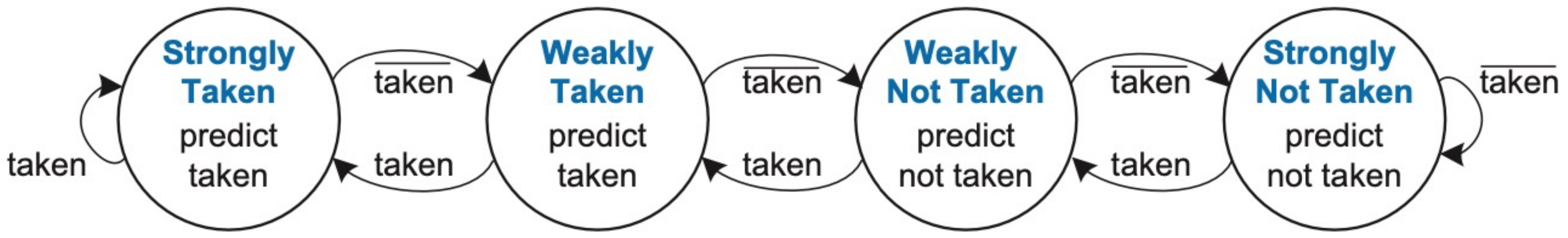
```
        addi s1, zero, 0   # s1 = sum = 0
        addi s0, zero, 0   # s0 = i = 0
        addi t0, zero, 10  # t0 = 10
for:
        bge  s0, t0, done  # i >= 10?
        add  s1, s1, s0    # sum = sum + i
        addi s0, s0, 1     # i = i + 1
        j    for           # repeat loop
done:
```

- One-bit predictor:

  Predicts correctly bge last time.

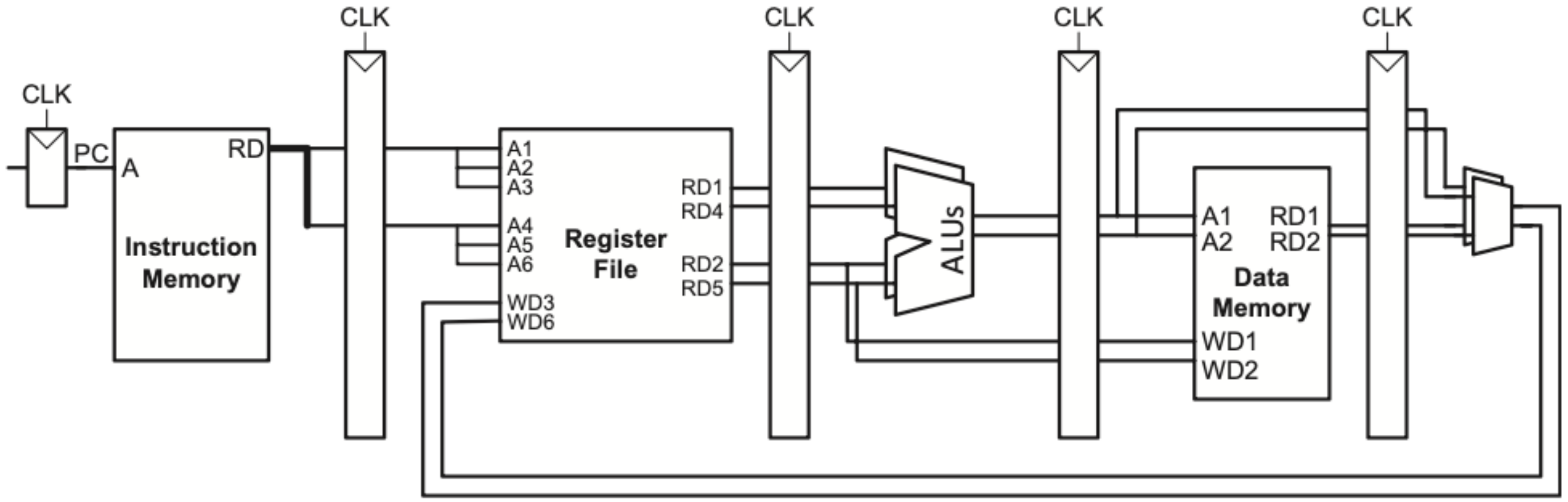
  Mispredicts j first and last time.

- Mispredicts first and last time of the loop.
- Overcome this with a two-bit predictor:


Strongly Taken predict taken
Weakly Taken predict taken
Weakly Not Taken predict not taken
Strongly Not Taken predict not taken
taken
$\overline{\text{taken}}$
taken
$\overline{\text{taken}}$
taken
$\overline{\text{taken}}$
taken
$\overline{\text{taken}}$

- Four states = two-bits to encode the states.
- Mispredicts only the last branch of a loop.

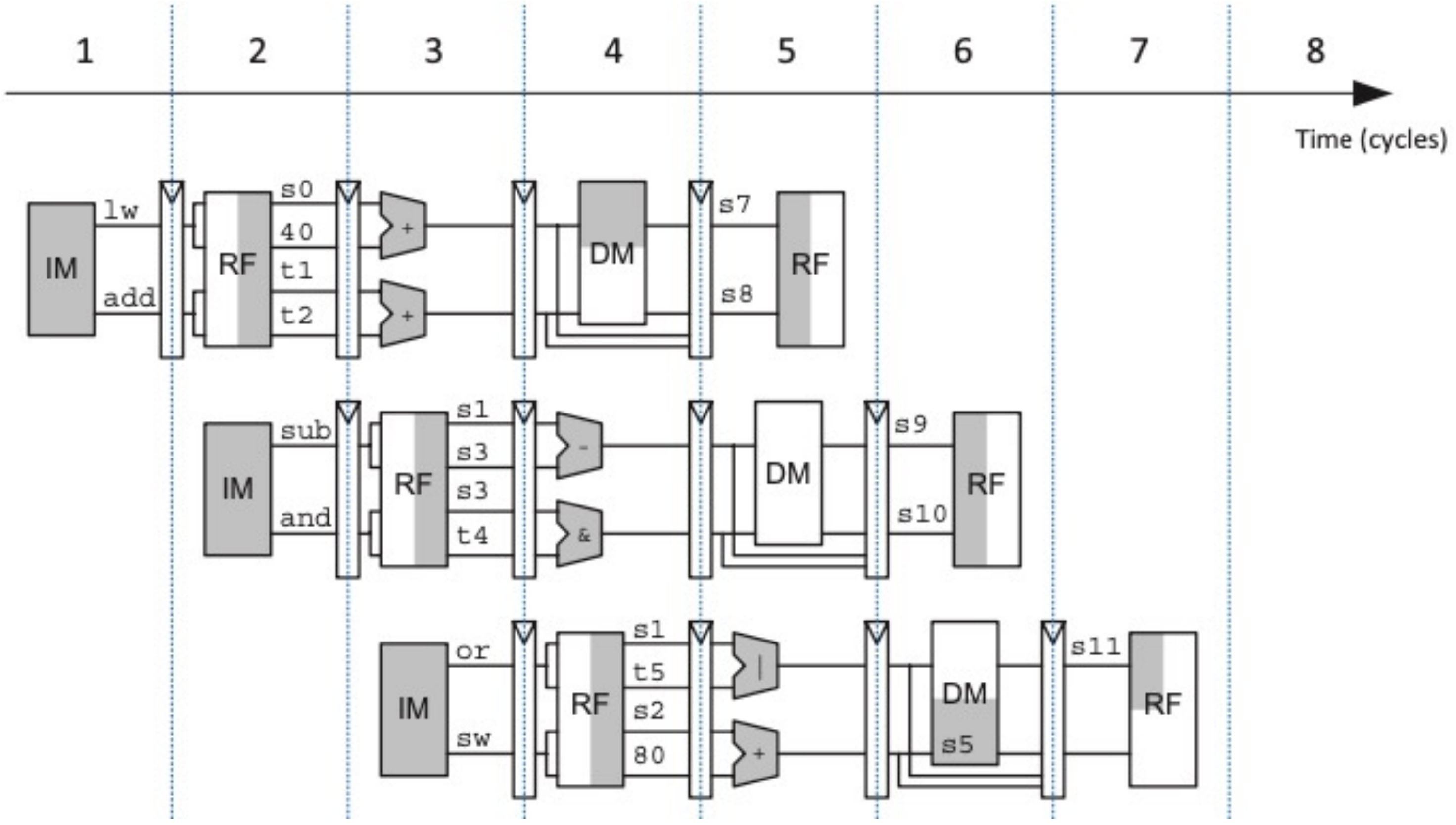
Superscalar Processor



- Two-way superscalar – execute TWO instructions on each cycle (CPI = 0.5, IPC = 2).
- Instruction memory – 2 read ports, i.e. fetch 2 instructions per cycle.
- Two copies of the ALU.
- Register file double number of ports (i.e. 4 read ports and 2 write ports).
- Data memory – two read ports and two write ports.
- Two instructions progress through CPU at the same time.

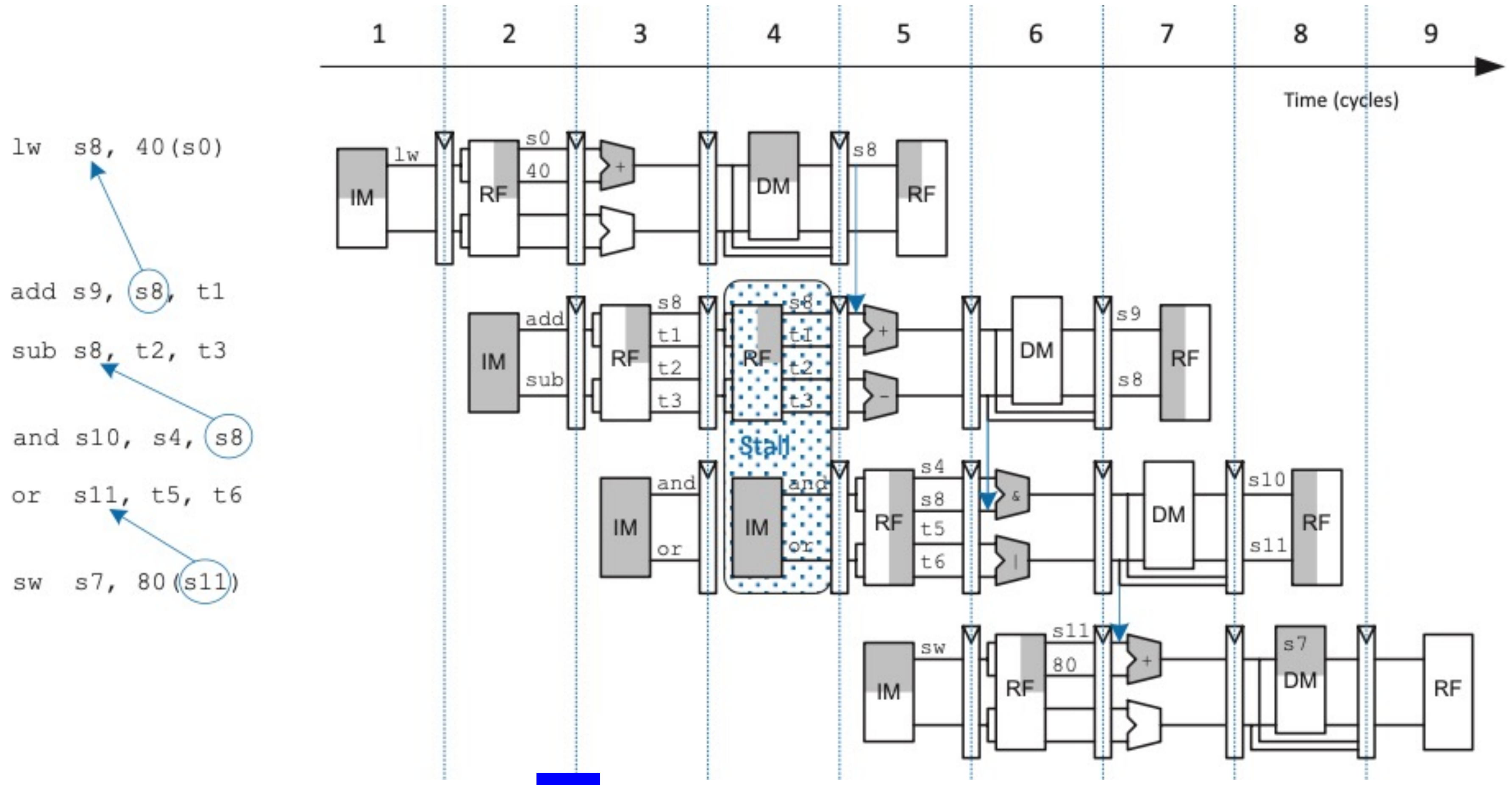
Superscalar Processor - Example

```
lw  s7, 40(s0)
add s8, t1, t2

sub s9, s1, s3
and s10, s3, t4

or  s11, s1, t5
sw  s5, 80(s2)
```



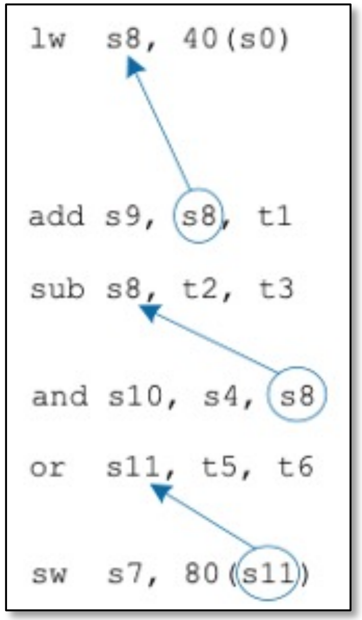
- Instruction per cycle = 2
- No data or control hazard in this code.

Superscalar Processor with data hazard

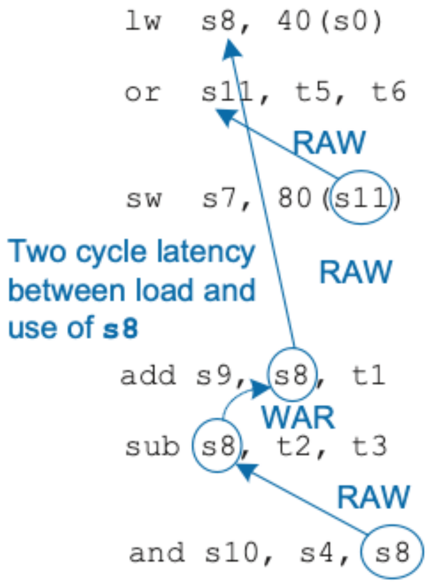


- Forwarding does not help add instruction – need to insert stall cycle, then forwarding.
- Other dependencies handled by forwarding. 5 cycles to issue 6 instructions: IPC = 1.2.

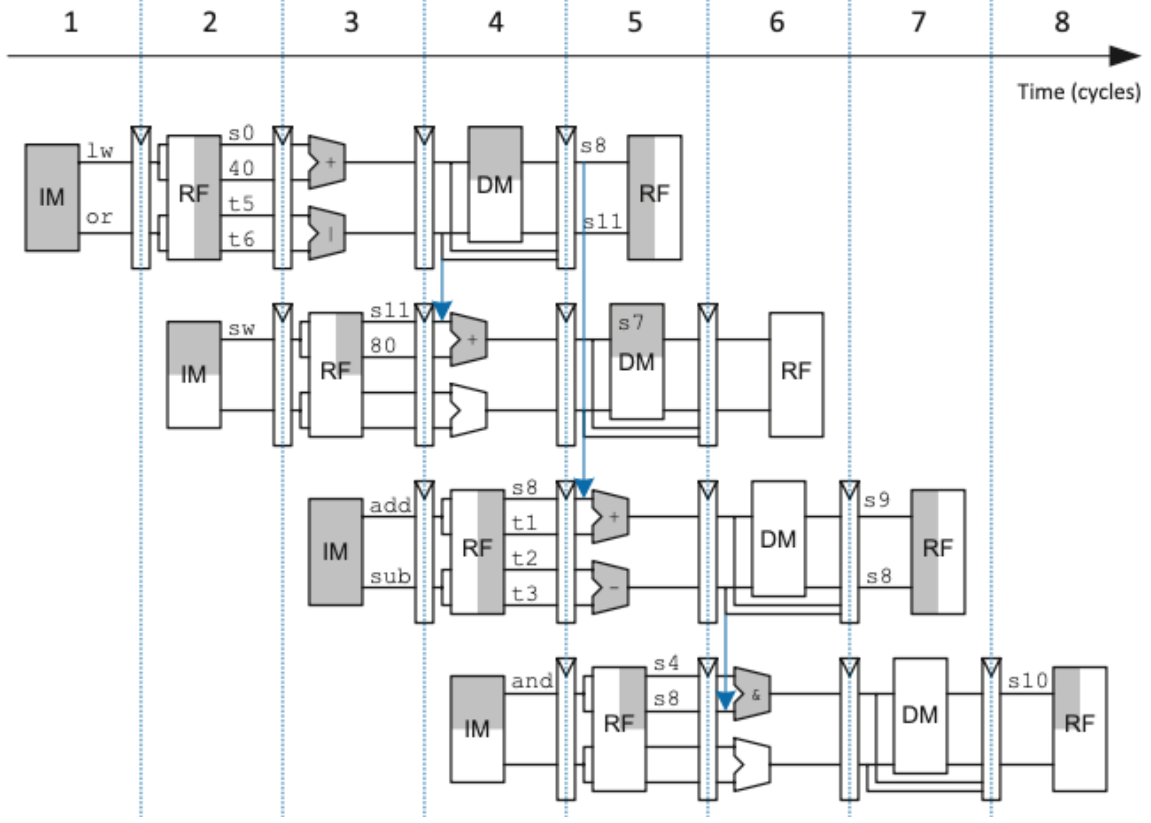
Out-of-Order Superscalar Processor (1)



```
lw   s8, 40(s0)

add s9, s8, t1
sub s8, t2, t3

and s10, s4, s8
or  s11, t5, t6

sw  s7, 80(s11)
```

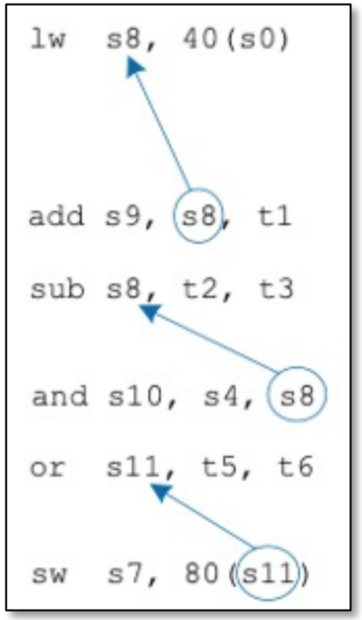


```
lw   s8, 40(s0)
or   s11, t5, t6
sw   s7, 80(s11)
add s9, s8, t1
sub s8, t2, t3
and s10, s4, s8
```

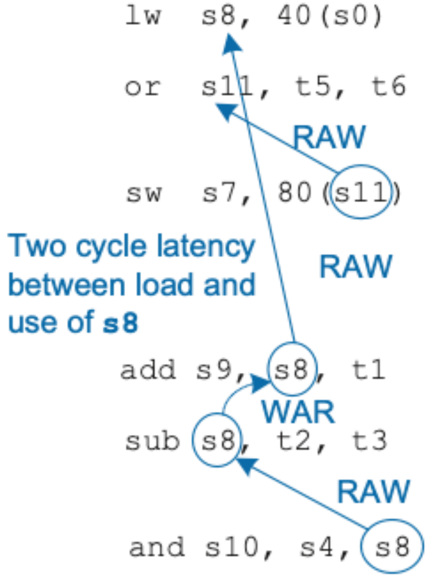


- Cycle 1: add, sub and and instructions use s8. Therefore, or instruction jumps ahead.
- Cycle 2: lw needs two cycle before data available. add can’t issue. sub use s8, cannot issue. Therefore, only sw can be issued because S11 can be forwarded.

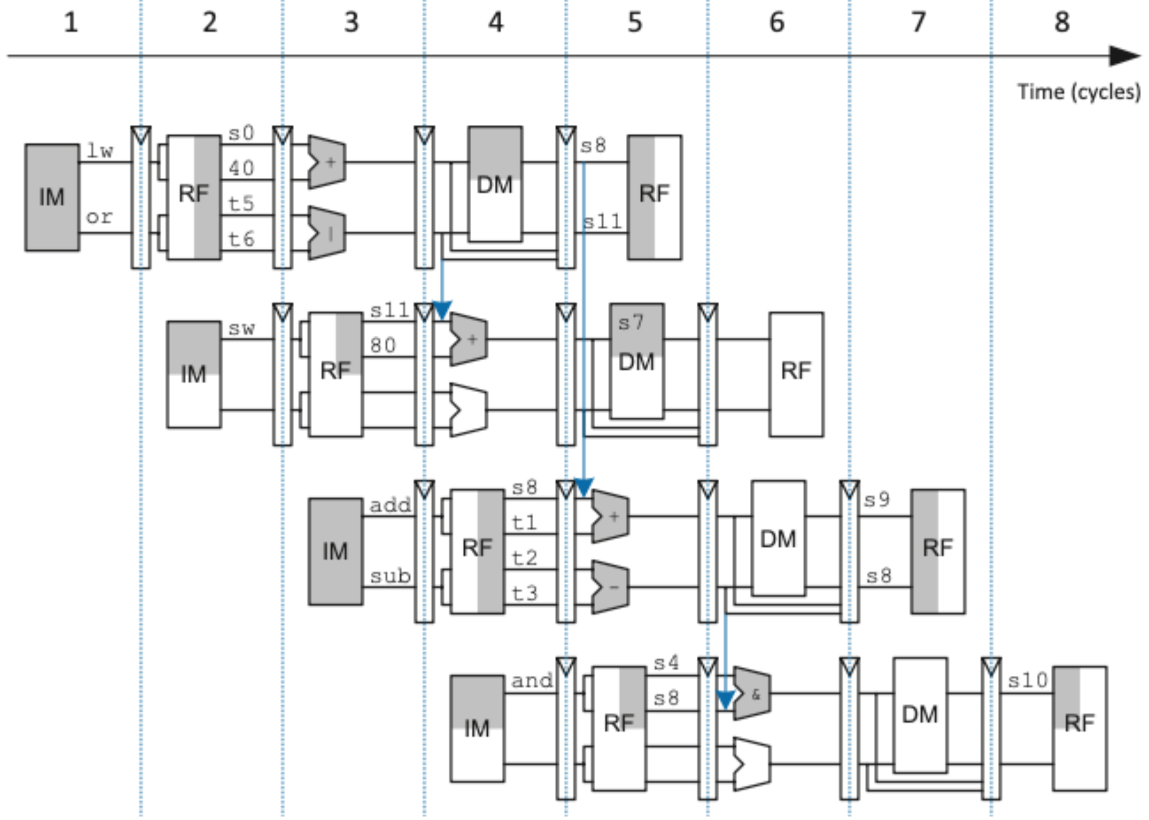
Out-of-Order Superscalar Processor (2)



```
lw   s8, 40(s0)

add  s9, s8, t1
sub  s8, t2, t3

and  s10, s4, s8
or   s11, t5, t6

sw   s7, 80(s11)
```



```
lw   s8, 40(s0)

or   s11, t5, t6

sw   s7, 80(s11)

add  s9, s8, t1
sub  s8, t2, t3

and  s10, s4, s8
```



- Cycle 3: Now add can be issued since s8 will be available, and sub can also go ahead.
- Cycle 4: The and can be issued.
- Six instructions in four cycles, IPC = 1.5 – better than 1.2 before.

Topics not covered by this module

1. **Computer arithmetics**
   - adders, multipliers, dividers
2. **Bus interface** (e.g. WishBone bus)
   - Interface with main memory, peripherals etc.
3. **Interrupt handling mechanism**
   - realtime applications, react to external events
4. **Stack and Heap**
   - Memory management in high-level languages

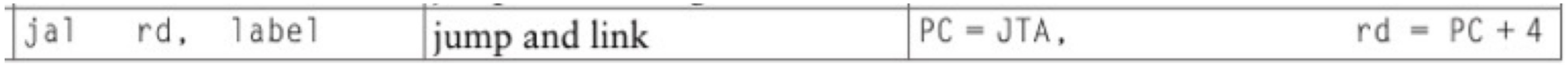
Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*"
by Sarah Harris and David Harris (H&H),

RISC-V Specific Omissions

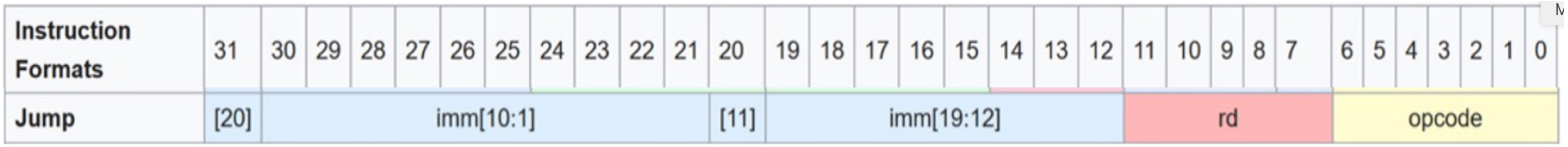
1. Control/Status Registers (CSRs)
2. Privileged mode vs User mode
3. Compressed instruction set (16-bit instructions)
4. Floating point architecture (64-bit)

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*"
by Sarah Harris and David Harris (H&H),

JAL instruction



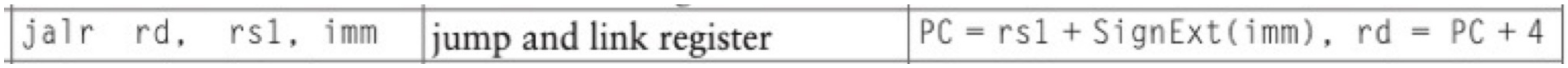
`jal rd, label`	jump and link	`PC = JTA, rd = PC + 4`



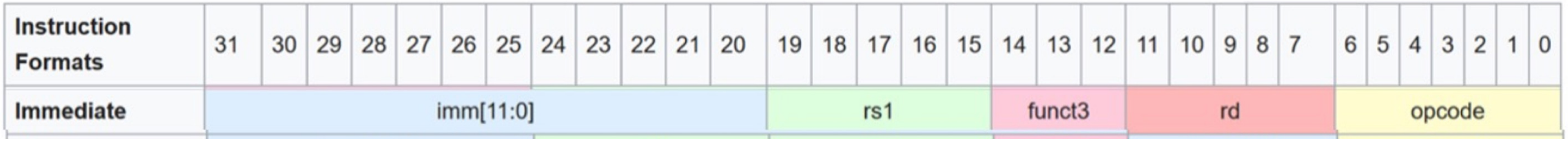
Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- **JAL** instruction is used for subroutine calls. (Used in the REF program.)
- **JTA** = Jump Target Address = **PC** value + signed immediate offset
- **PC** is loaded with the JTA
- **rd** = return address = **PC + 4**, i.e. address of next instruction
- Note that the format of the immediate value is unusual. Bit 0 is always 0. In other word, offset is always an even number

JALR instruction



jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4



Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	imm[11:0]												rs1					funct3			rd					opcode						

- JALR instruction is also used for subroutine calls, but different from JAL.
- **JTA = rs1 + SignExt(imm)**, i.e. derived from source register **rs1**
- Note that the immediate offset is only 12 bit and it is sign-extended to 32-bits before adding to **rs1**
- Finally, **rd** stores the return address
- SPECIAL CASE, **JALR zero, 0(ra)** or **JALR x0, 0(x1) = RET**