Imperial College
London

# Lecture 11

# Additional Topics

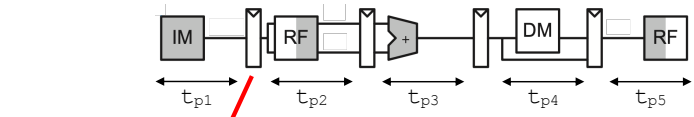
Peter Cheung
Imperial College London

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

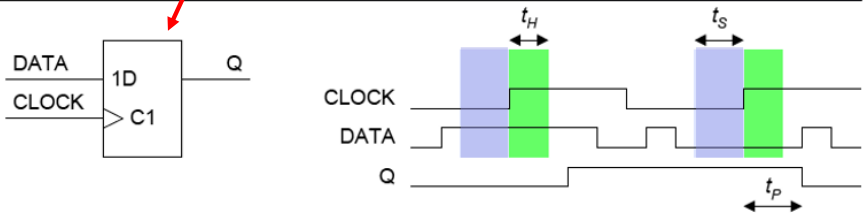


This lecture is about how virtual memory is managed in a process and a computer system.

## 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**: $\frac{1}{\max(t_{p1},t_{p2},t_{p3},t_{p4},t_{p5})+t_s}$

PYKC 25 Nov 2025 | EIE2 Instruction Architectures & Compilers | Lecture 11 Slide 2

So far, we have only considered three-stage pipeline architecture.

Consider the maximum clock frequency such a pipelined CPU can achieve.

Each stage involves two time constraints: 1) propagation delay of the logic $t_p$, 2) the setup time of the clocked circuit (i.e. register or D-FF).

A D-FF only works correctly if the input data is STABLE some time BEFORE the active clock edge. The minimum time that data MUST BE stable before the clock edge is known as setup time $t_s$ (shown in BLUE).

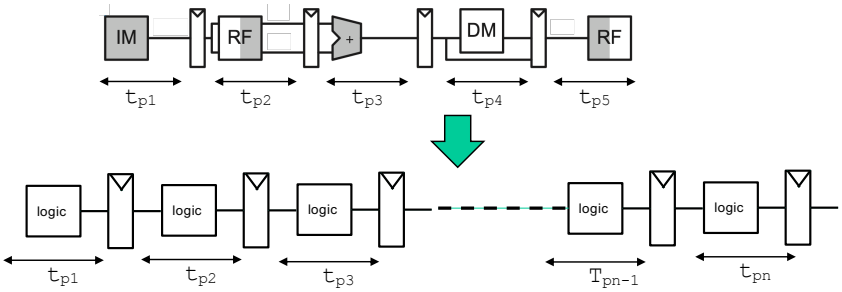
Data must also stay STABLE some time AFTER the active clock edge. This is known as hold time $t_h$ (shown in GREEN).

If data changes within the setup and hold time window, the output of the D-FF is unpredictable. This will cause a crash in the computer if it happens.

Therefore, the minimum period between two active clock edges is the worst-case delay in the logic = $\max(t_{p1}, t_{p2}, t_{p3}, t_{p4}, t_{p5}) + t_s$. Hold time places not part in this consideration.

In other words, to increase the clock speed of a processor, we can attempt to reduce the worst-case propagation delays between pipeline registers. One way to achieve this is add more pipeline stages.

## 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.



This idea leads the technique known as deep pipelining. Here logic stages are splitted into multiple stages with pipeline registers inserted in between. The best strategy is to design the CPU such that the delay between registers are nearly equal in all stages, so that the clock frequency is not dominated by the worst stage in the pipeline.

Increasing the number of stages in a pipeline theoretically will not affect the performance in terms of Cycle per Instruction (CPI). It remains slightly higher than 1 due to stalling and flushing as a result of data and control hazards. In an earlier example, we stated that CPI could be around 1.25 – taking an average of 1.25 clock cycles to execute one instruction.

Overall performance increase comes from the increase clock frequency due to reduced cycle time $T_c$. Instruction time = $CPI \times T_c$.

Why can we not keep increasing the pipeline stages? Two reasons: 1) deeper pipeline results in more complicated logic to detect and mitigate hazards; 2) more hazards will be generated due to data dependency and branches, increasing the CPI. There is a trade off between reducing $T_c$ resulting in increased CPI due to hazards.

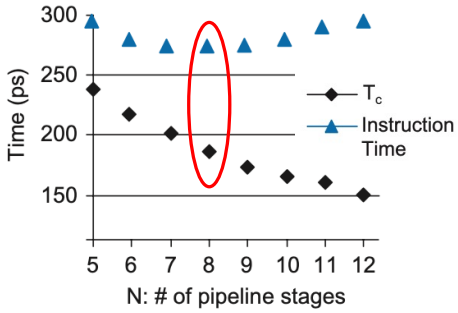
Further, as propagation in the logic decrease due to increased pipeline stages, the setup time of the flip-flops dominates. This results in diminishing return.

As an example, a modern ARM processor as used in most mobile devices uses a 13-stage pipeline architecture.

## 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),



Let us consider a concrete example.

Assume that your single-cycle RISC-V processor has a propagation delay of 750ps overall. You may be considering what performance gain you might obtain by introducing N stages of pipeline.

The assume here are as stated in the slide above.

The question – how many pipeline stages should one use for best performance?

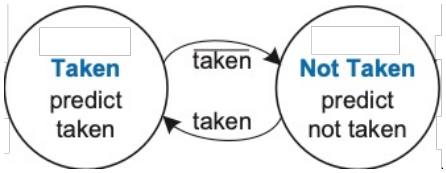
Given that the register setup time (overhead) is 90ns. If we split the 750ps into N stages with each delay, $T_c = \frac{750}{N} + 90$ ns.

We are told that minimum stages should be 5, and for this, CPI is 1.25. Every additional stage will add, on average, another 0.1 CPI. Therefore, CPI of N-stage processor = 1.25 + 0.1 (N-5) for N ≥5.

The graphs here plots the number of stages vs $T_c$. It can seen from the plot that the minimum instruction time is when N=8 and $T_c \times CPI = 281$ns. This is only a marginal improvement from the 5-stage architecture, which has an instruction time of 295 ps. Make N larger increases the instruction time and reduces the performance.

## 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),



So far, we always fetch the **next instruction** even if the current instruction is a branch or jump instruction. This is the cause of control hazard in a pipelined architecture. This is the same as predicting that a branch is NEVER TAKEN.

We can improve the performance of the processor by reducing the number of branch hazard encountered by making better prediction on whether a branch instruction is taken. For example, lots of program loops are written with the conditional branch instruction at the end of the loop branching BACKWARDS. These loop structure suggests that a branch destination addrss that goes backwards is normally taken and should be predicted as such. In contrast, forward branches or jumps are usually NOT taken. This strategy results in a **static branch prediction** depending on the location of the branch destination address. It requires little hardware and does better than NOT having any prediction at all.

However, programs can be more complex than this. Some backward branches are normally NOT taken while forward branches are, depending on how codes are generated and the type of loops that are being executed.

A better strategy is to assume that if the branch was taken last time, it is predicted to be taken next time as well. This requires the processor to establish a table of branch instructions with a 1-bit indicator that starts the most recent result (taken or not taken). This effective is a two-state state machine.

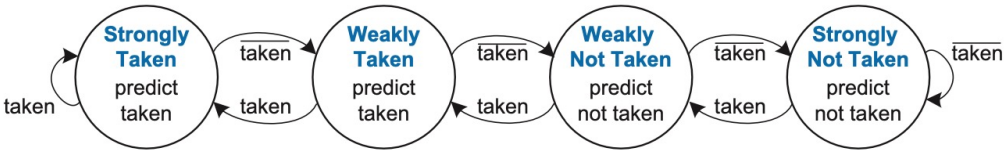
This table is known as a branch target buffer, and it also stores the destination address for fast execution.

## 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:
```

10 times

- 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:




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

PYKC 25 Nov 2025 EIE2 Instruction Architectures & Compilers Lecture 11 Slide 6

Consider the code segment in the slide.

This is a loop that goes around 10 times. With one-bit predictor, the bge correctly predict not taken until the last time around the loop. However, if the code is executed again, the first time would also be mispredicted.

The j instruction is mispredicted both the first and last time.

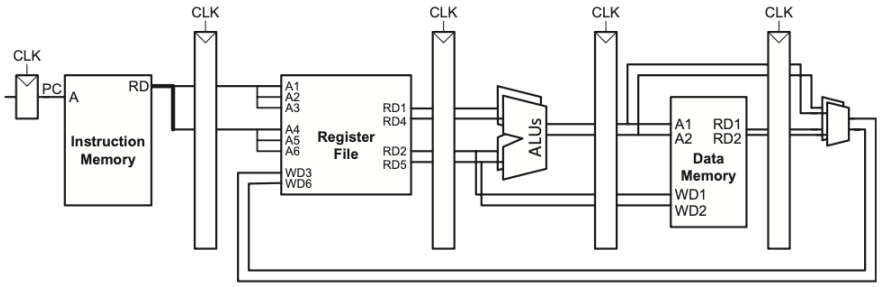
In general, one-bit predictor mispredict branches the first and last time around a loop.

This can be improved by using a slightly more sophisticated predictor – a two-bit predictor giving four states: strongly taken, weakly taken, weakly not-taken and strongly not-taken.

If a branch is taken two or more times, the predictor is in the strongly taken state. If a branch is not taken twice or more, the predictor is in the strongly not-taken state.

For the code segment in the slide, the bge instruction is in the strongly not-taken state. It predicts incorrectly first time the loop is exited (branch to done:), but the predictor stays in the weakly not-taken state. Therefore, next time when the code is executed again, it returns to the strongly not-taken state. In general, the two-bit predictor only gets it wrong once around the loop, not twice.

## 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.

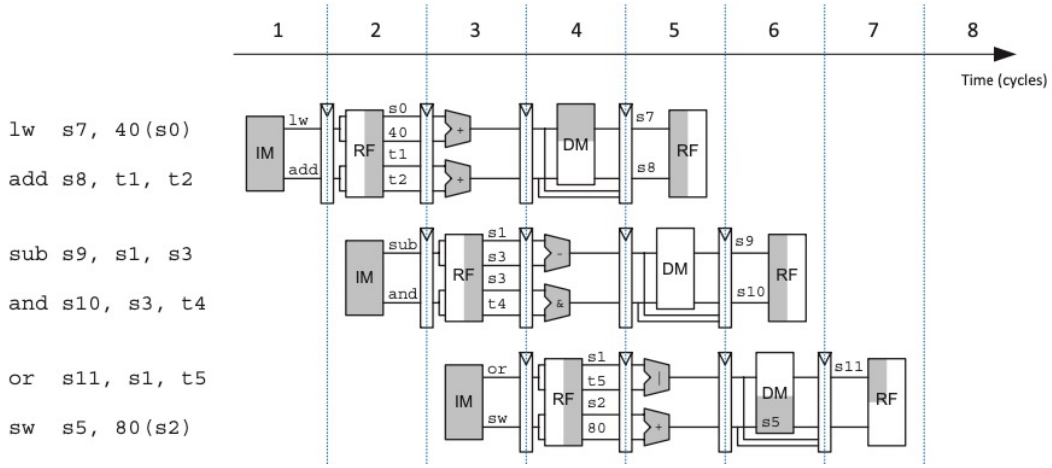


A simple pipeline processor has a CPI that is 1 or above (due to hazard mitigation). Superscalar processor, on the other hand, can reduce this to lower than 1 by duplicating hardware within a processor.

Show here is a two-way superscalar processor with two ALU working in parallel. To keep both ALU busy, two instructions needs to be issue at each clock cycle. This also means that the Register File needs to double its ports and do does the data memory and other internal hardware.

For example, modern processor such as Intel i9 can handle 6 instructions per cycle and ARM Cortex A78 processor executes up to 3 instructions per cycle.

## Superscalar Processor - Example



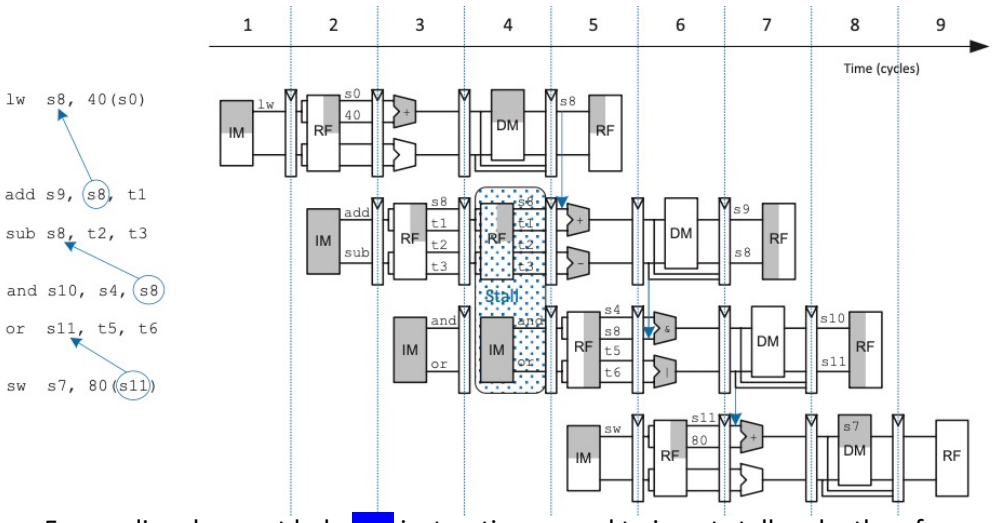

- Instruction per cycle = 2
- No data or control hazard in this code.



Here is an example of our 5-stage pipeline processor with two-way superscalar executing six instructions in 3 cycles, giving a CPI of 0.5.

Note that the instruction sequence here does not generate any data or control hazards. This is in general NOT the case.

## 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.

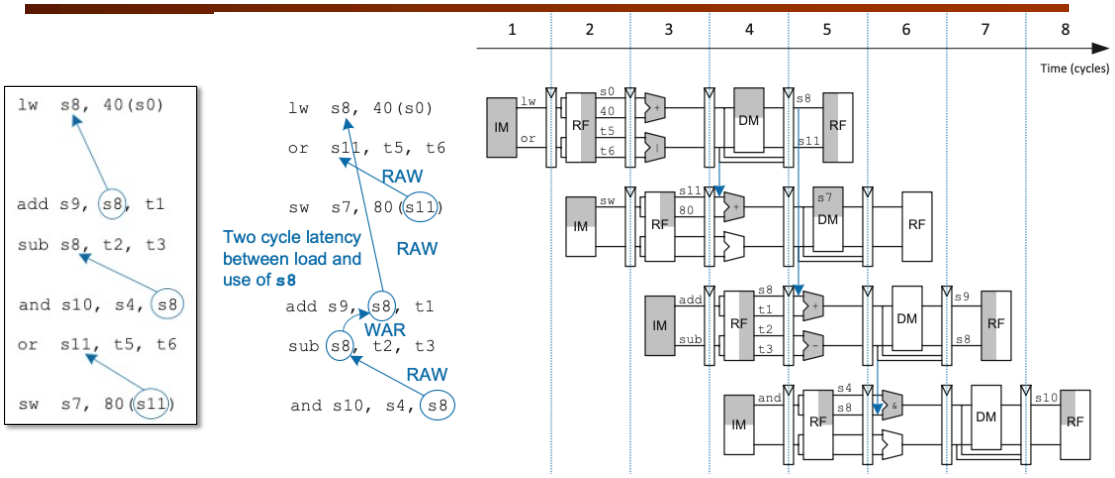


This code snippet has plenty of data hazard.

The `lw` instruction cannot provide values for s8 until cycle 5, but the `add` instruction needs it in cycle 4. Therefore there is no solution except to delay issue of the `add` instruction to cycle 2, and to insert a stall cycle in cycle 4. There are other dependencies such as the `and` instruction also requires s8, and the `sw` instruction requires s11. However, the stall cycle will allow these hazard to be resolved through forwarding after the stall cycle is added.

However, inserting the stall cycle results in 6 instructions executed in 5 cycle, a IPC of 1.2, much lower than the ideal case of IPC = 2.0.

## Out-of-Order Superscalar Processor (1)




- 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.



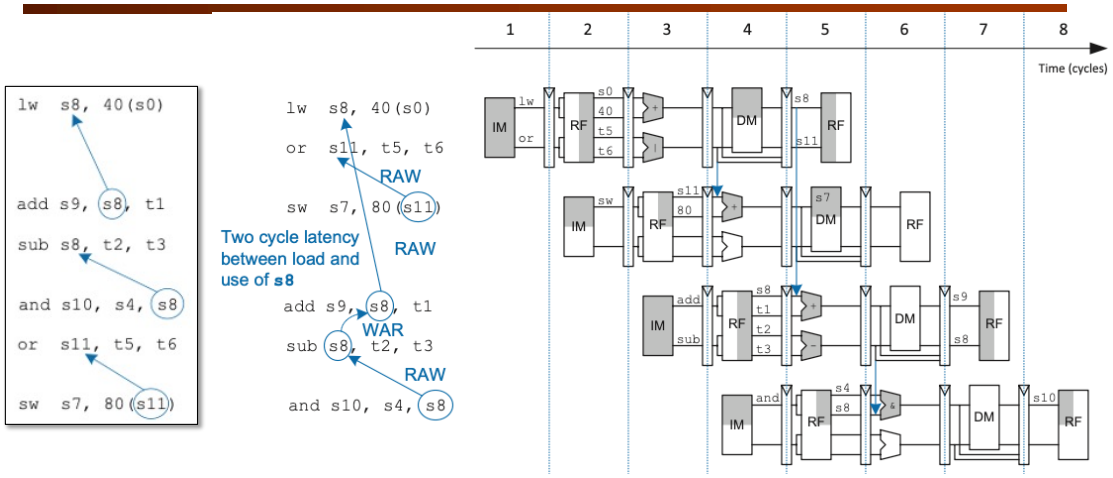
To mitigate this, all modern superscalar processor introduce Out-of-Order execution. Here the processor fetch a number of instructions from program memory into a buffer. This instructions are analyzed for data dependency and issue not in sequence, but in a way that hazard due to data dependency is avoided.

As an example, the same code snippet is executed in the order shown in the slide above.

In cycle 1, `add` `sub` `and` all depend on S8, which is not available. Therefore, `or` is issued with `lw`.

In cycle 2, s8 is still not available, so `sw` is the only instruction that can be executed and is therefore issued on its own because it relies on s11, which is ready by the time the `or` has done its job.

## Out-of-Order Superscalar Processor (2)




- 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.



Finally, the remaining three instructions are issued in the following two cycles as shown.

This means that the 6 instructions is now executed in four cycles, give us a IPC of 1.5 – better than 1.2 before.

Note that in the slide, the data hazards are labelled as RAW and WAR.

RAW stands for Read-after-Write hazard. The hazard is cause by `or` instruction writing to s11, and it is read by the following instruction `sw`. Similar the dependency of `add` on `lw` because of s8 is another RAW hazard.

WAR stands for Write-after-Read hazard. The example is with `add` on `sub` because s8 is used by both instructions. `sub` must not write to s8 before `add` finishes reading s8.

These hazards can be overcome by a technique known as **Register Renaming**. This is a topic that will be left to the Advanced Computer Architecture elective in the 3rd year.

## 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),



This short one-term module only allows 8 teaching weeks. Not everything that should be in the module are covered due to shortage of time. Here are 5 topics that I wish I have time to include, but not able.

## 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),



There are also a number of interesting aspects of the RISC-V processor that you may like to read yourself in your spare time.

# 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

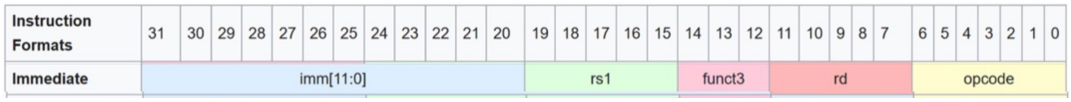


.

Answer some questions from a number of students about JAL and JALR.

# JALR instruction

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




- 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**