Architectures of Computer Systems (BIE-APS), Lecture 4

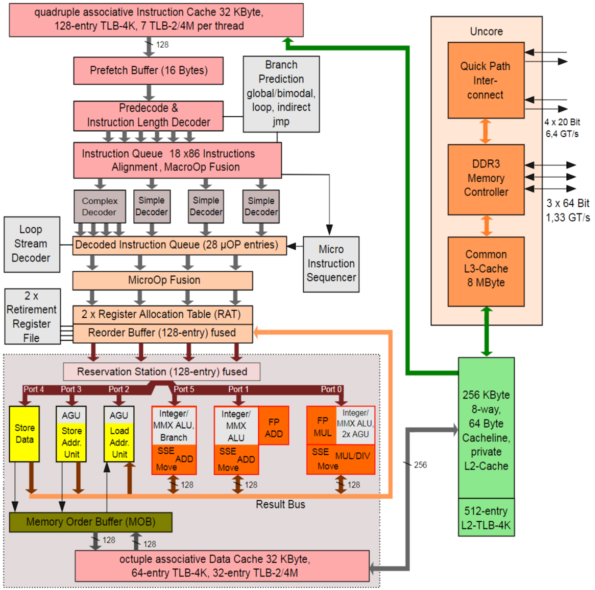
Pipelined RISC microarchitecture

Michal Štepanovský, Pavel Tvrdík

Czech Technical University in Prague
Faculty of Information Technology
`https://courses.fit.cvut.cz/BIE-APS`

2022
(Document version: 1. 11. 2022 16:46)

Motivation – Intel Nehalem (Core i7)

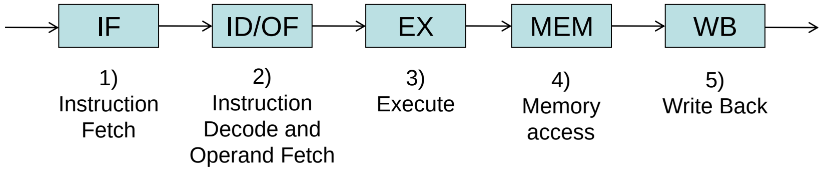


quadruple associative Instruction Cache 32 KByte, 128-entry TLB-4K, 7 TLB-2/4M per thread
128
Prefetch Buffer (16 Bytes)
Branch Prediction global/bimodal, loop, indirect jmp
Predecode & Instruction Length Decoder
Instruction Queue 18 x86 Instructions Alignment, MacroOp Fusion
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Loop Stream Decoder
Decoded Instruction Queue (28 µOP entries)
Micro Instruction Sequencer
MicroOp Fusion
2 x Retirement Register File
2 x Register Allocation Table (RAT)
Reorder Buffer (128-entry) fused
Reservation Station (128-entry) fused
Port 4
Port 3
Port 2
Port 5
Port 1
Port 0
Store Data
AGU Store Addr. Unit
AGU Load Addr. Unit
Integer/ MMX ALU, Branch
SSE ADD Move
Integer/ MMX ALU
FP ADD
SSE ADD Move
FP MUL
Integer/ MMX ALU, 2x AGU
SSE MUL/DIV Move
128
128
128
Result Bus
Memory Order Buffer (MOB)
128
128
octuple associative Data Cache 32 KByte, 64-entry TLB-4K, 32-entry TLB-2/4M
256
Uncore
Quick Path Inter-connect
4 x 20 Bit 6,4 GT/s
DDR3 Memory Controller
3 x 64 Bit 1,33 GT/s
Common L3-Cache 8 MByte
256 KByte 8-way, 64 Byte Cacheline, private L2-Cache
512-entry L2-TLB-4K

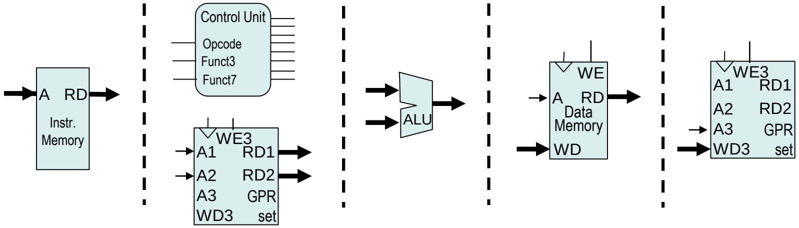
http://en.wikipedia.org/wiki/File:Intel_Nehalem_arch.svg

Instruction Pipelined Microarchitecture

Consider RISC ISA(3,0) and decompose instruction execution into 5 stages:



The decomposition of an instruction cycle into more stages is called **instruction pipelining** (hereafter **IP**).



Instruction Pipelining I

Stage **IF** (Instruction Fetch) is skipped in the following diagrams because it is identical for all instructions.

Pipelined processing of instruction `lw rd,imm(rs1)`:

2) ID/OF: fetch data from register `rs1`.
3) EX: add with `imm` to get a memory address.
4) MEM: read the memory cell.
5) WB: write the obtained value to register `rd`.

Pipelined processing of instruction `sw rs2,imm(rs1)`:

2) ID/OF: fetch data from registers `rs1` and `rs2`.
3) EX: add the value from `rs1` with `imm` to get a memory address.
4) MEM: write the data from register `rs2` to the memory cell.
5) WB: –.

Instruction Pipelining II

Pipelined processing of instruction `add rd,rs1,rs2`:

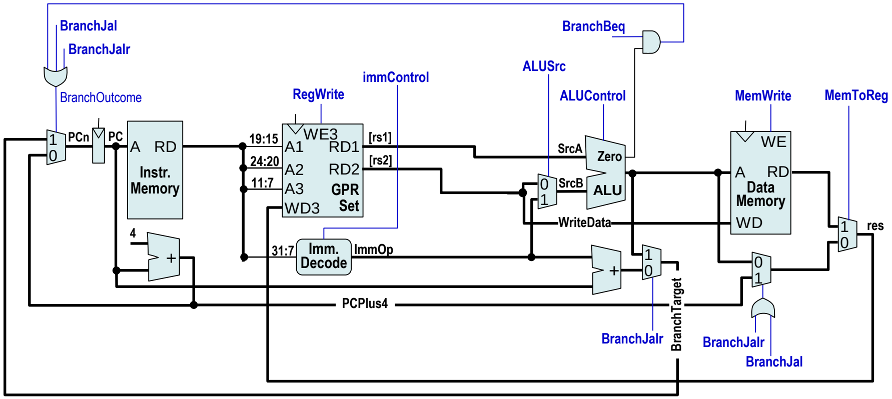
2) ID/OF: fetch data from registers `rs1` and `rs2`.
3) EX: sum up these values in ALU.
4) MEM: –.
5) WB: write the sum to register `rd`.

Pipelined processing of instruction `beq rs1,rs2,imm`:

2) ID/OF: fetch data from registers `rs1` and `rs2`.
3) EX: generate the Zero signal using the diff of these two values and compute the branch target address.
4) MEM: if Zero $= 1$, load the PC register with the branch target address, otherwise continue at the sequentially next instruction address.
5) WB: –.

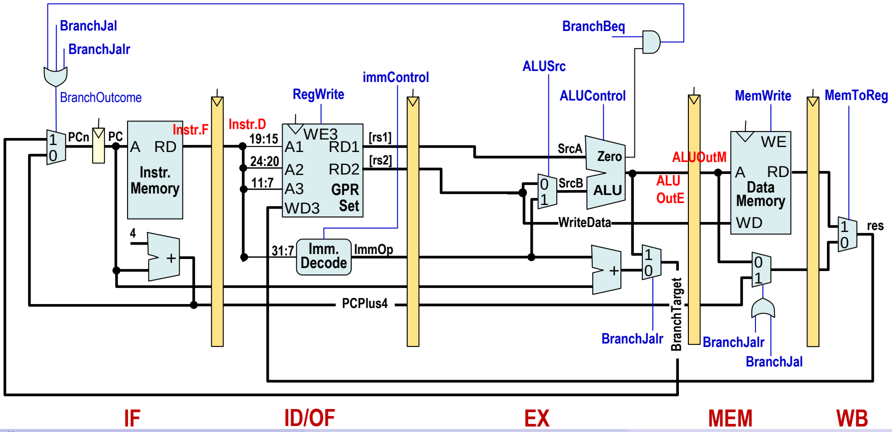
picoRISC-V: Single-Cycle Microarchitecture

Recall the scheme of the microarchitecture from Lecture 3, Slide 39:



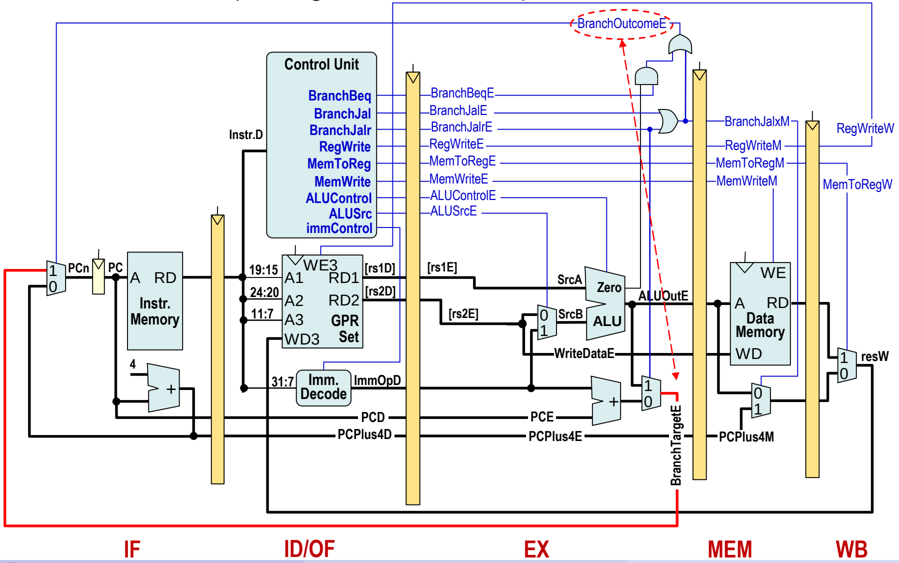
picoRISC-V: IP Requires Interstage Registers I

- Inserting **interstage registers** splits the IP into several stages. Data and control signals of the IP are transferred to the next stage at the **rising edge** of the clock, otherwise are **stored** in the interstage registers. For example, data bus `ALUOut` has different data in stage EX (`ALUOutE`) and in stage MEM (`ALUOutM`).



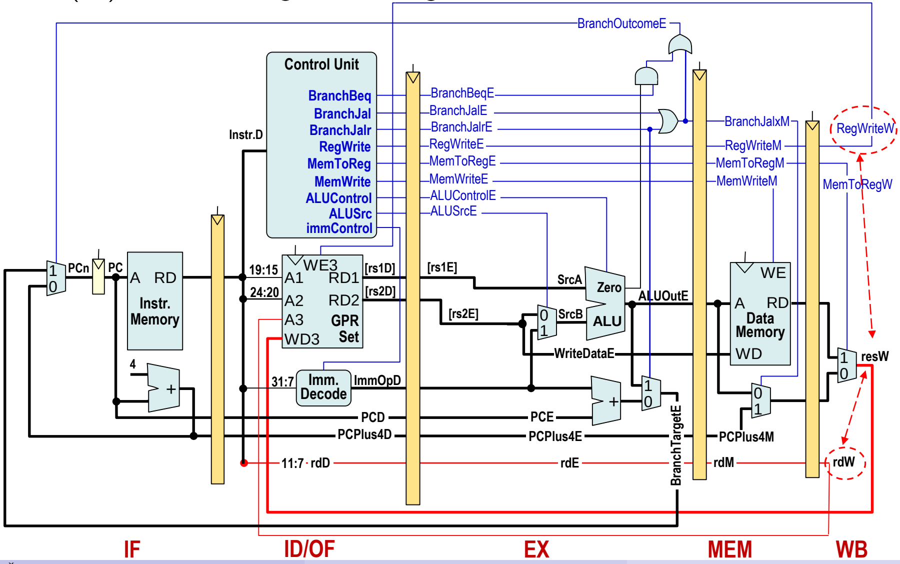
picoRISC-V: IP Requires Interstage Registers II

- The same applies to control signals. They must be pipelined synchronously with the corresponding data in the data paths.



picoRISC-V: IP Requires Interstage Registers III

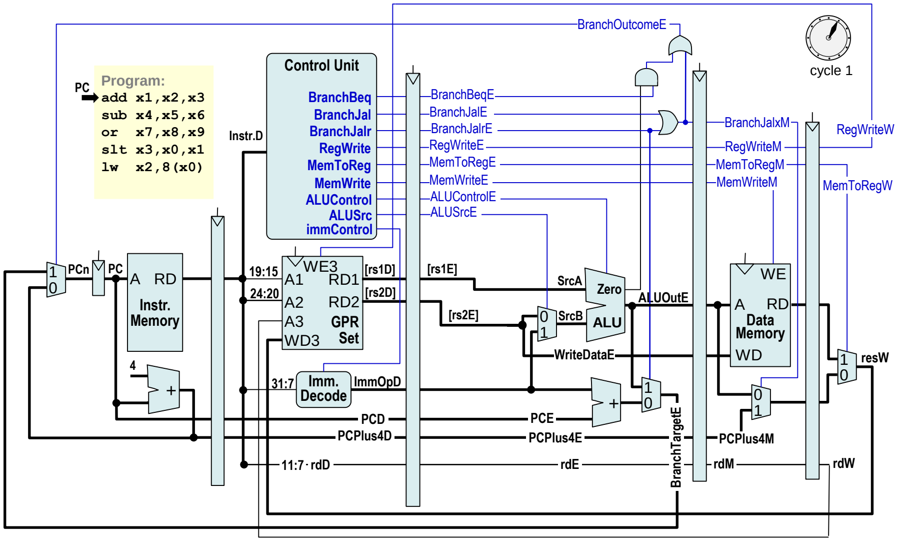
- The necessary correction of a data path carrying the destination register (`rd`) number through the IP stages.



Pipelined Instruction Processing: Example

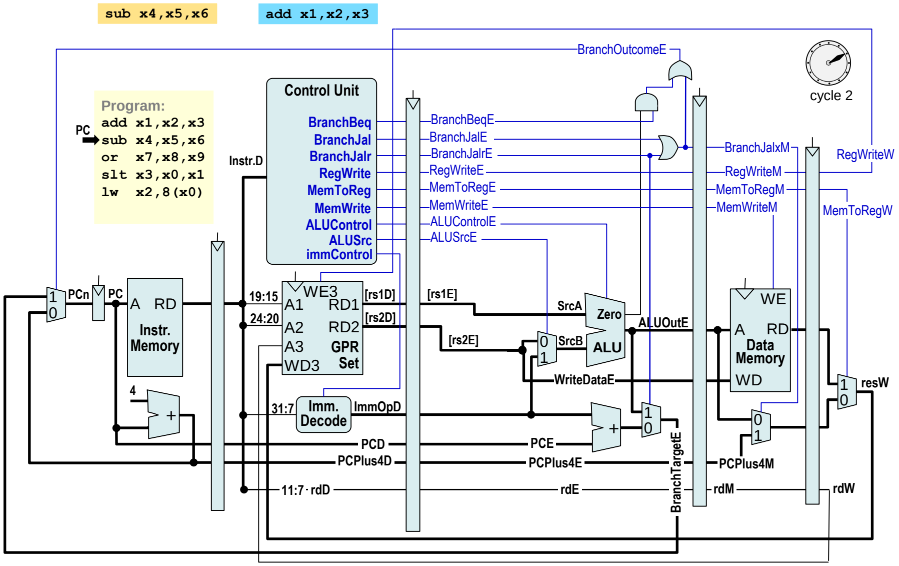
- Clock cycle 1: The 1st program instruction `add x1,x2,x3` is fetched from the instr. memory addressed by [PC].

`add x1,x2,x3`



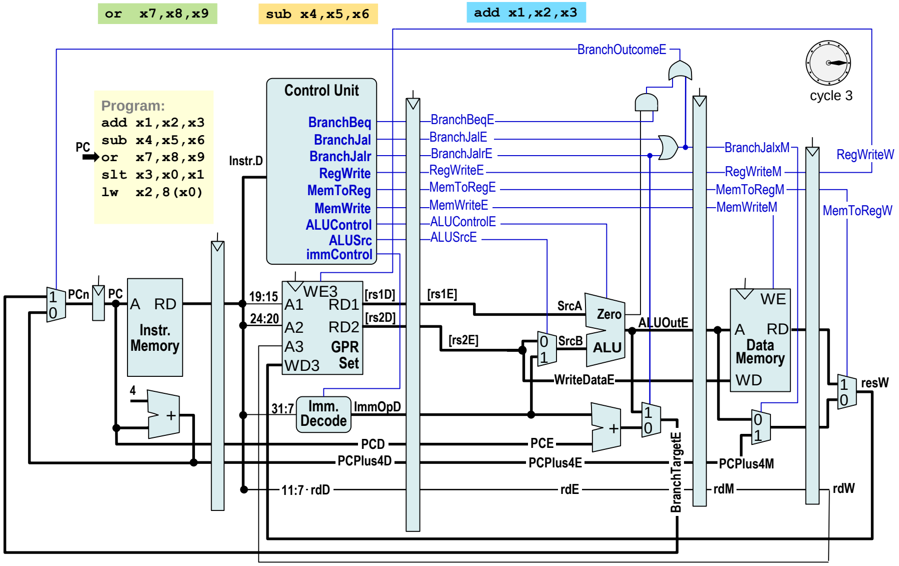
Pipelined Instruction Processing: Example

- Clock cycle 2: instruction `add` moves to the ID stage, source registers `x2,x3` are fetched & instruction `sub x4,x5,x6` is fetched from the instr. memory.



Pipelined Instruction Processing: Example

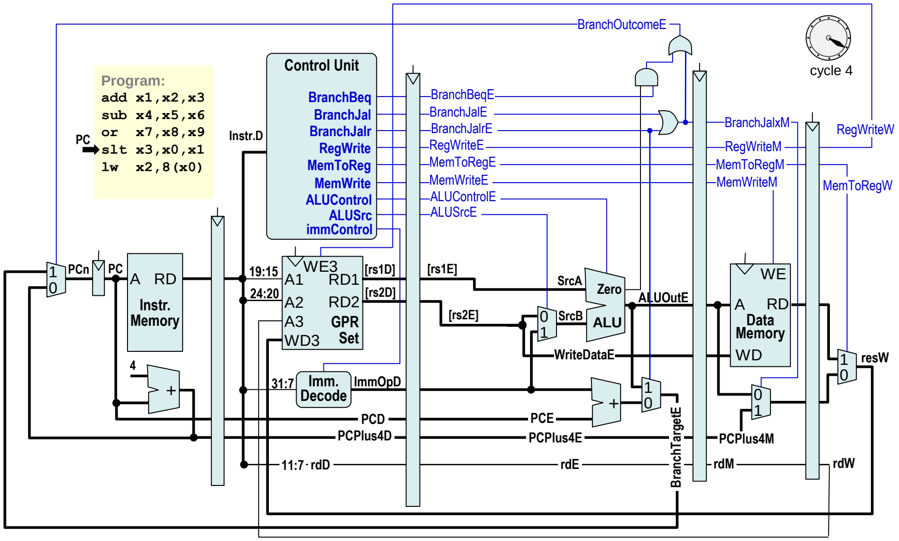
- Clock cycle 3: instruction `add` is executed in the EX stage & source registers `x5,x6` are fetched & instruction `or x7,x8,x9` is fetched.



Pipelined Instruction Processing: Example

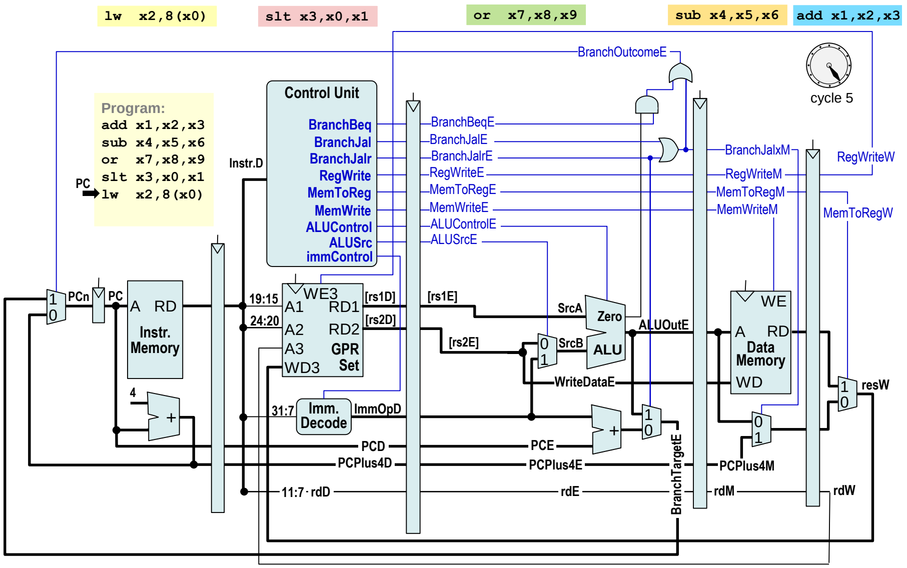
- Clock cycle 4: The value `[x2]+[x3]` bypasses the data memory in stage MEM & ALU computes `[x4]-[x5]` & source registers `x8,x9` are fetched.

`slt x3,x0,x1`   `or x7,x8,x9`   `sub x4,x5,x6`   `add x1,x2,x3`



Pipelined Instruction Processing: Example

- Clock cycle 5: the value `[x2]+[x3]` is written to register `x1`.



Instruction Pipelining Parallelism – Ideal Scenario

	1	2	3	4	5	6	7	8	9	10
IF	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10
ID/OF		I1	I2	I3	I4	I5	I6	I7	I8	I9
EX			I1	I2	I3	I4	I5	I6	I7	I8
MEM				I1	I2	I3	I4	I5	I6	I7
WB					I1	I2	I3	I4	I5	I6
	$\xleftrightarrow{5\tau}$					$\xleftrightarrow{\tau}$	$\xleftrightarrow{\tau}$	$\xleftrightarrow{\tau}$	$\xleftrightarrow{\tau}$	$\xleftrightarrow{\tau}$

Let $\tau = max\{\tau_i\}_{i=1}^{k}$, where τ_i is the **propagation delay** of stage i in a k-stage IP. Assume an ideal program execution (no stalls).

- Time to process n instructions in a k-stage IP:

$$T_{k,n} = k\tau + (n-1)\tau.$$

- Ideal speedup wrt single-cycle microarchitecture from Lecture 4:

$$S_{k,n} = \frac{nT(instr)}{T_{k,n}} = \frac{nk\tau}{k\tau + (n-1)\tau}, \quad \lim_{n\to\infty} S_{k,n} = k.$$

Problems of Instruction Pipelining Parallelism = Hazards

- IP does not reduce CPI of individual instructions, just the opposite. Recall the speed vs. throughput discussion in Lecture 1.
- Since multiple instructions are processed simultaneously, requirements for **shared resources** can result in conflicts. This is called **hazards**.
- A shared resource is a resource that is repeatedly used in various stages of IP.
- **Hazard types**:
  - **data** (caused by data dependencies: RAW, WAR, WAW),
  - **control** (caused by instructions that change PC),
  - **structural** (the number of simultaneous requirements for a given resource exceeds the number of its instances).
- Hazard prevention can result in IP **stalls** or IP **flushes**.
- A deeper IP (more stages) results in shorter sequences of gates in each stage, which enables to increase the clock frequency of the processor, but more stages imply higher overhead (demand to better arrange instructions into a pipeline and longer delays in case of stalls or flushes).

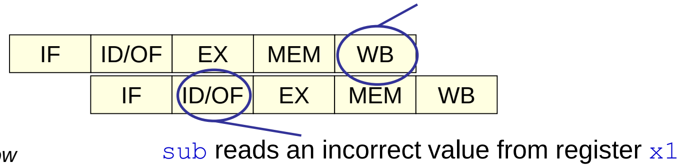
Unresolved Hazards Change Program Semantics

Data hazard:

I1. `add x1,x2,x3`

I2. `sub x4,x1,x3`

sequential instruction flow



add writes a new value to register x1
IF
ID/OF
EX
MEM
WB
IF
ID/OF
EX
MEM
WB
sub reads an incorrect value from register x1

Control hazard:

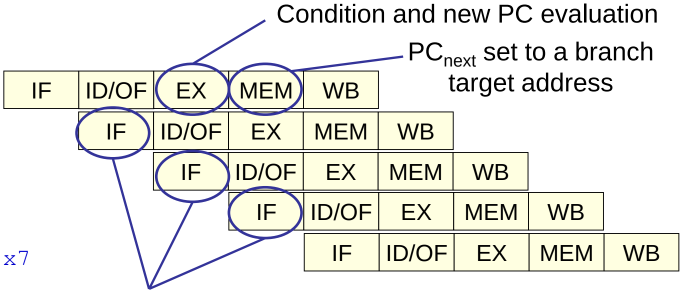
I1. `beq x3,x7,M1`

I2. `add x6,x1,x2`

I3. …

I4. …

I5. `M1: add x4,x6,x7`

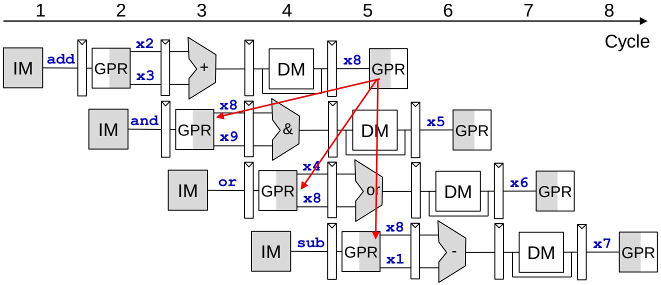


Condition and new PC evaluation
PC$_{next}$ set to a branch target address
IF
ID/OF
EX
MEM
WB
IF
ID/OF
EX
MEM
WB
IF
ID/OF
EX
MEM
WB
IF
ID/OF
EX
MEM
WB
IF
ID/OF
EX
MEM
WB

Should instructions I1, I2 and I3 be fetched (and processed then)?

picoRISC-V: Cause of Data Hazards and their Resolution

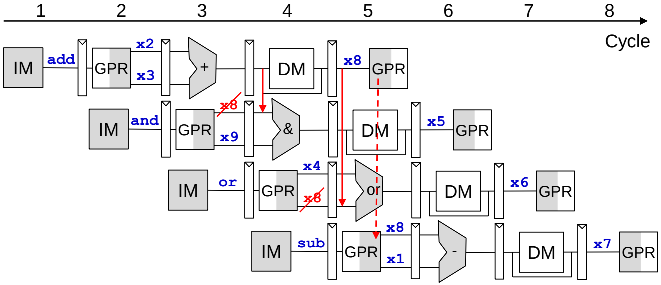
```
add x8, x2, x3

and x5, x8, x9

or  x6, x4, x8

sub x7, x8, x1
```



- The GPR set is accessed in the WB stage and in the ID/OF stage:
  - WB: in the **first** half of the clock cycle, data are written into a register.
  - ID/OF: in the **second** half of the clock cycle these data can be read from the register.
- The example illustrates 2 RAW hazards. Instructions `and` and `or` need to read register `x8` **before** the preceding instruction `add` writes the requested value into `x8`.
- Instruction `sub` is OK, it gets `[x8]` just in time.
- RAW hazards can be **resolved** without slowing down the IP by **forwarding**.

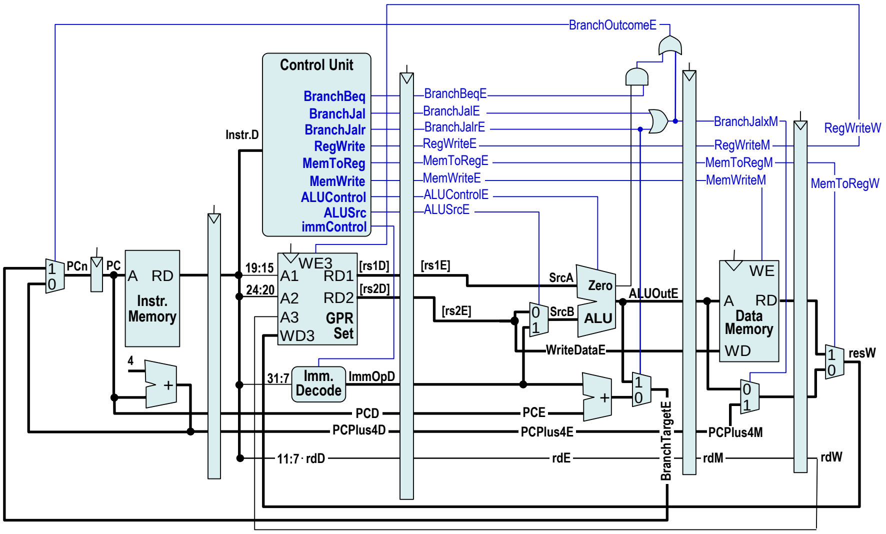
picoRISC-V: Data Hazards Resolved by Forwarding I

```
add x8, x2, x3

and x5, x8, x9

or  x6, x4, x8

sub x7, x8, x1
```

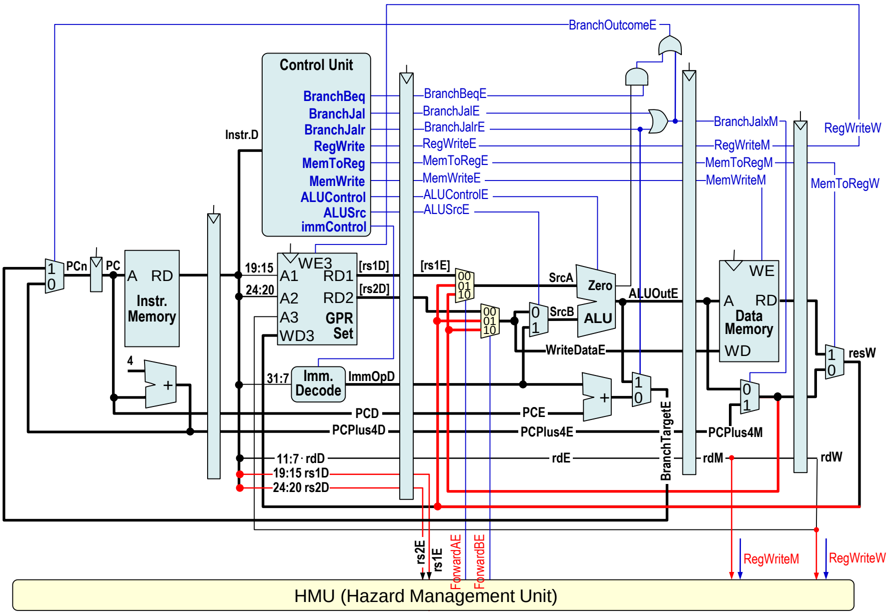


- A RAW hazard appears if the destination register in the instruction in stage MEM or WB (`RegWrite` = 1) equals to a source register in the EX stage.
- Resolving RAW hazards with Forwarding is based on the fact that
  - the data to be read from the source register in stage EX **is already available** in stage MEM or WB on data paths and
  - therefore it can be taken from those data paths **before** writing it into the destination register.
- So: a subsequent data dependent instruction will NOT get the data from its source register but from the IP data path.

picoRISC-V: The Current Microarchitecture



picoRISC-V: Data Hazards Resolved by Forwarding II



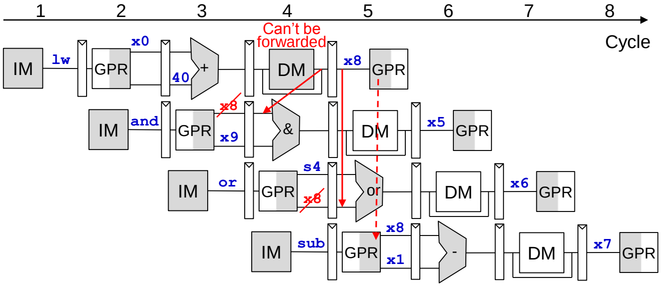
picoRISC-V: Data Hazards Resolved by Forwarding III

HMU has the following inputs and outputs:

- Inputs `RegWriteM` and `RegWriteW` are control signals indicating whether the data on data paths `ALUOutM` and `ALUOutW` in stages MEM and WB, respectively, are to be written into the GPR set.
- Inputs `rdM` and `rdW` are 5bit numbers of the destination register in stages MEM and WB, respectively.
- Input `ALUControlE` is the ALU opcode of the instruction in stage EX.
- Inputs `rs1E` and `rs2E` are 5bit numbers of the source registers in stage EX.
- Output `ForwardAE` is the control signal indicating whether the data on data path `ALUOutM` or `ALUOutW` in stages MEM or WB are to be forwarded to the input `SrcA` of the ALU and it happens only if `ALUControlE` is an ALU op code and `rs1E` = `rdM` or `rs1E` = `rdW`.
- Output `ForwardBE` is a control signal indicating whether the data on data path `ALUOutM` or `ALUOutW` in stages MEM or WB are to be forwarded to the input `SrcB` of the ALU and it happens only if `ALUControlE` is an ALU op code and `rs2E` = `rdM` or `rs2E` = `rdW`.

picoRISC-V: Data Hazards Resolved by IP Stalls

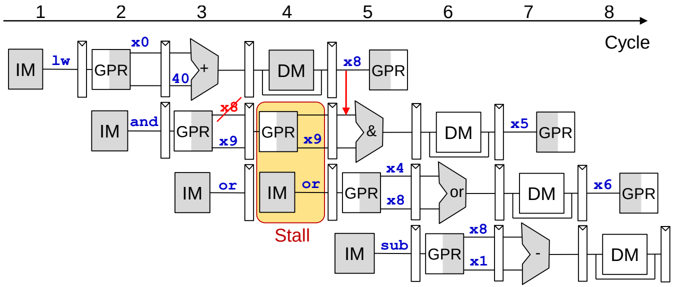
```
lw  x8, 40(x0)

and x5, x8, x9

or  x6, x4, x8

sub x7, x8, x1
```



- If an instruction in stage ID/OF needs to fetch a source register **before** its valid contents is really **available** on IP data paths, then stage ID/OF must be **stalled** (see instruction and above).
- In general, stage ID/OF is **stalled** (= **frozen**) **until** the data is loaded from memory in stage MEM.
- Once the data appear on a data path, it is **forwarded** using the technology on Slide 21 to minimize the stall penalty.
- Instruction stalls worsen the IP throughput (IPC) and increase instruction latencies (CPI).

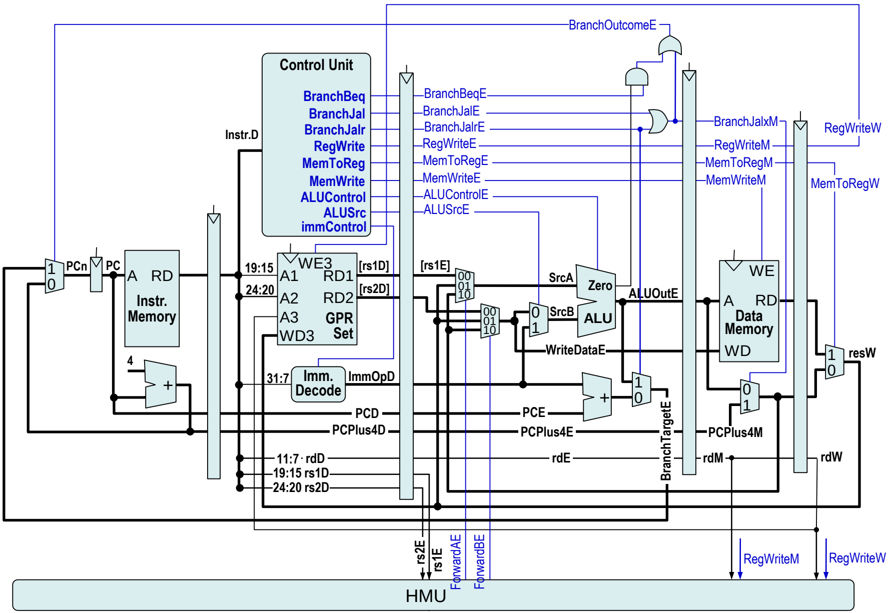
picoRISC-V: Data Hazards Resolved by IP Stalls

```
lw   x8, 40(x0)

and x5, x8, x9

or  x6, x4, x8

sub x7, x8, x1
```

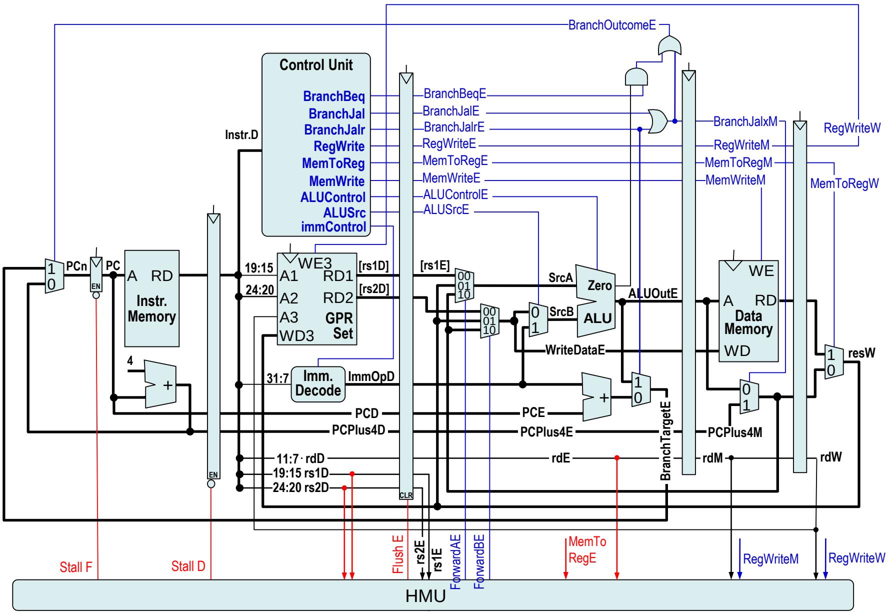


- A stall of IP requires to perform simultaneously 2 actions with interstage registers using their two control inputs EN and CLR:
  1. **freezing the contents of two left interstage registers**. It is implemented by blocking their write-enable inputs EN by signals StallF and StallD.
  2. **inserting instruction** nop (= **bubble**) into stage EX so that the rest of the IP can proceed. It is implemented by activating control input CLR of the **interstage register between stages ID/OF and EX** with signal FlushE.
- In general, several bubbles can flow in the IP simultaneously.

picoRISC-V: The Current Microarchitecture



picoRISC-V: Data Hazards Resolved by Stalls – Implementation



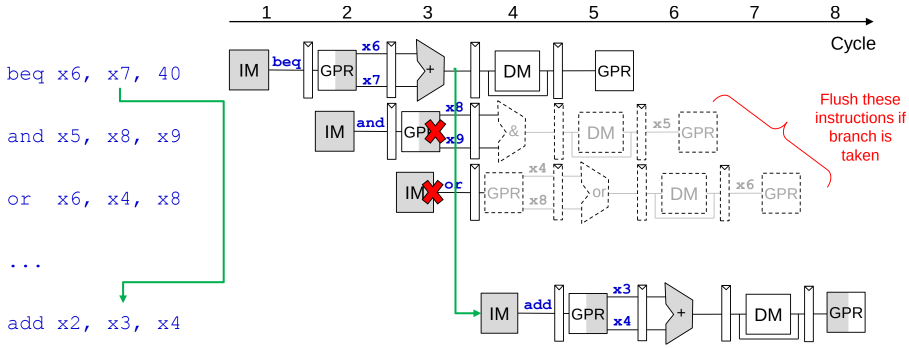
picoRISC-V: Data hazards Resolved by Stalls – Implementation

Therefore, HMU functionality is augmented as follows:

- If `rdE` matches `rs1D` or `rs2D` and `ALUControlD` indicates an ALU op in stage ID/OF and `MemToRegE` indicates a `lw` instruction in stage EX, then HMU detects the hazard from Slide 23,
  except that the hazard is detected when instruction `lw` is in stage EX, i.e., before it enters stage MEM.
- Then HMU sets
  - output control signal `FlushE` to reset the interstage register between stages ID/OF and EX, which is equivalent to inserting a **bubble** into stage EX.
  - output control signals `StallF` and `StallD` to freeze stages IF and ID/OF.
- Therefore, instruction `lw` proceeds to stage MEM, while the subsequent ALU instruction remains frozen in stage ID/OF.
- In the next cycle, the whole IP moves forward and therefore the data `resW` loaded from the memory are forwarded to `SrcA/SrcB`, exactly as on Slide 21 (in parallel with writing to register `rdW`).

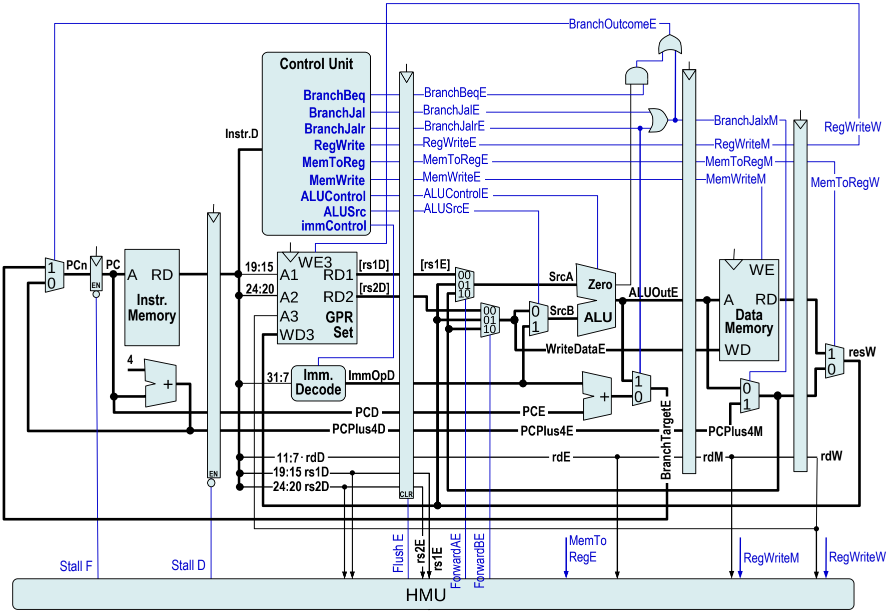
picoRISC-V: Control Hazards by Instruction beq

```
beq x6, x7, 40

and x5, x8, x9

or  x6, x4, x8

...

add x2, x3, x4
```

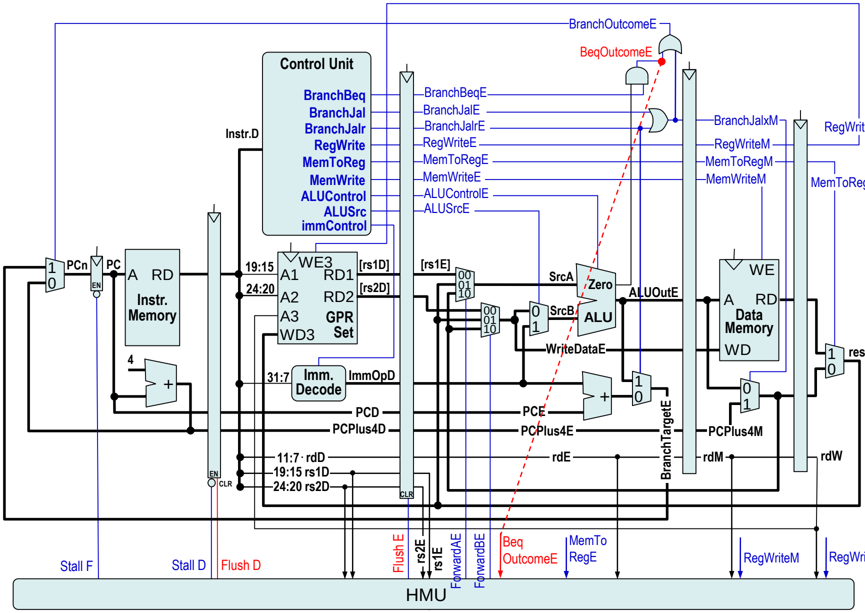


- Whether two registers are equal or not ($Zero = 1$) is not known until stage EX.
- In the mean time, IP has started sequential processing of 2 further instructions.
- If $Zero = 1$, then these instructions must be canceled and this again translates into insertion of **2 bubbles** (by activating FlushD and FlushE)).

picoRISC-V: The Current Microarchitecture

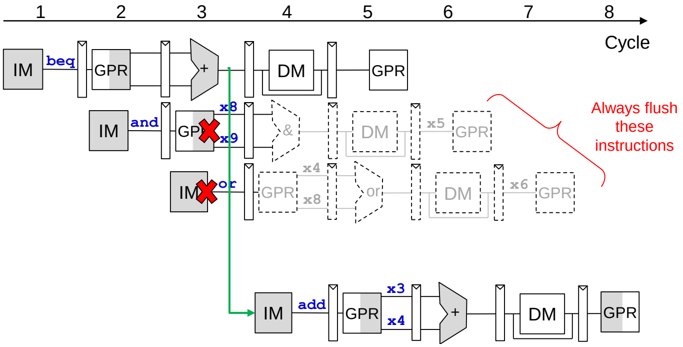


picoRISC-V: Control Hazards Resolved by Flushing



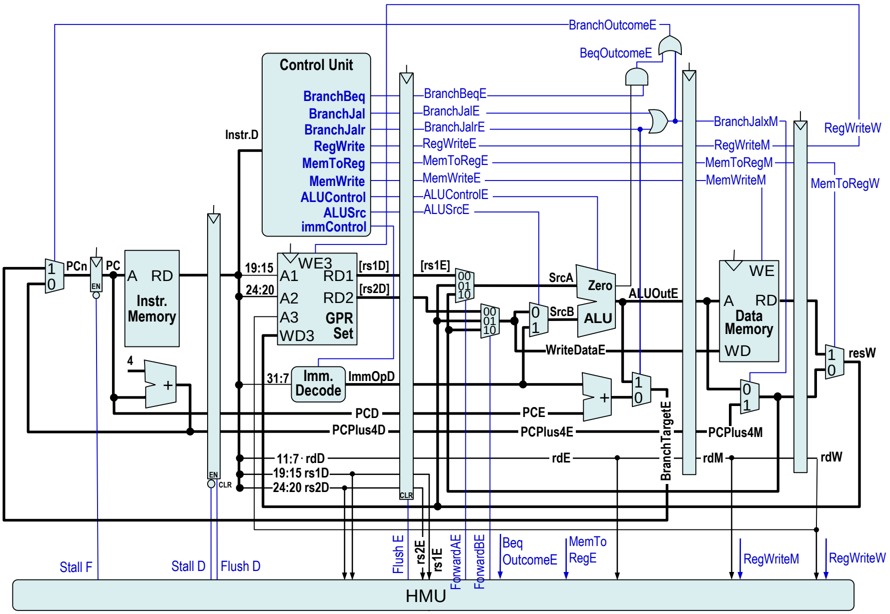
picoRISC-V: Control Hazards by Instructions jal and jalr

```
jal x1, 40

and x5, x8, x9

or  x6, x4, x8

...

add x2, x3, x4
```

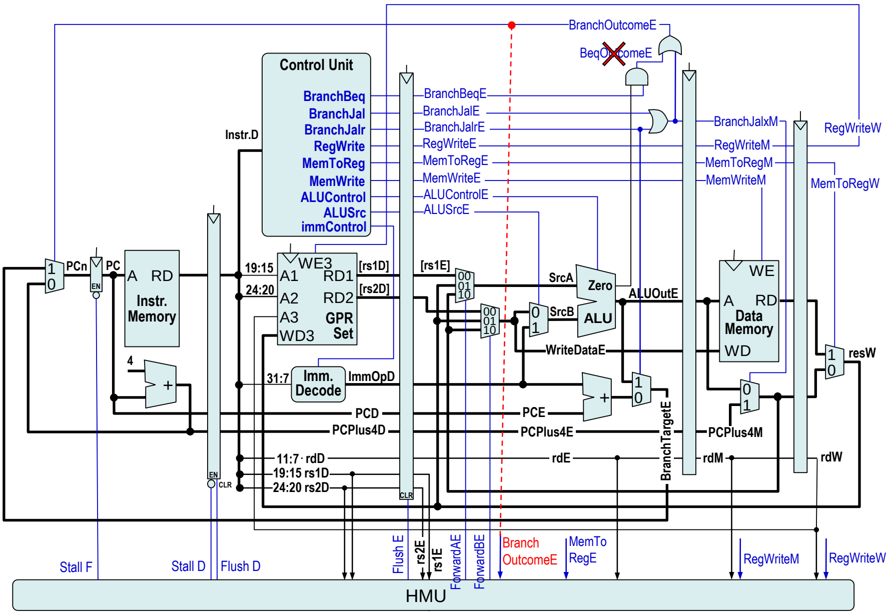


- Both jal and jalr set the new value to PC within phase EX.
- This means that two sequentially next instructions are being processed.
- However, these instructions **must be canceled**.
- **Hazard resolution**: Similarly as for the beq instruction except that the branch is unconditional. Just wait until instruction jal or jalr gets to stage EX (detected by control signals BranchJalE and BranchJalrE) and then activate FlushD and FlushE.

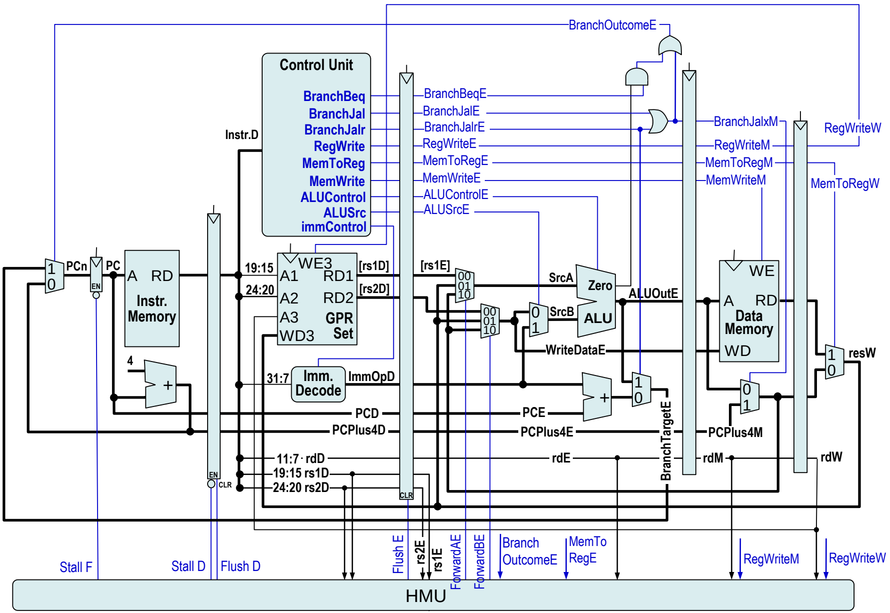
picoRISC-V: The Current Microarchitecture



picoRISC-V: Control Hazards by Instructions `jal` and `jalr`



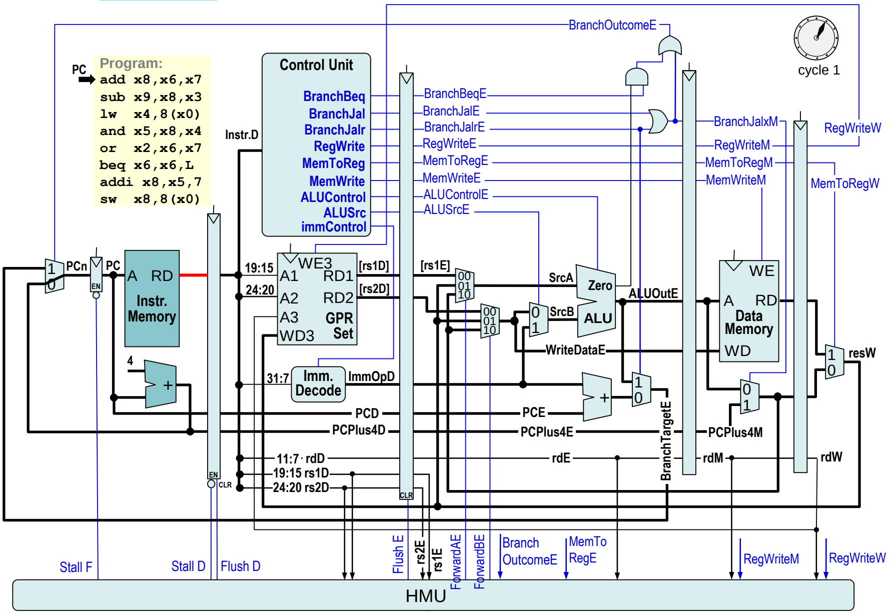
The Final Design of Pipelined picoRISC-V Microarchitecture



Example of Program Execution I

`add x8,x6,x7`

Program:
```
add  x8,x6,x7
sub  x9,x8,x3
lw   x4,8(x0)
and  x5,x8,x4
or   x2,x6,x7
beq  x6,x6,L
addi  x8,x5,7
sw   x8,8(x0)
```

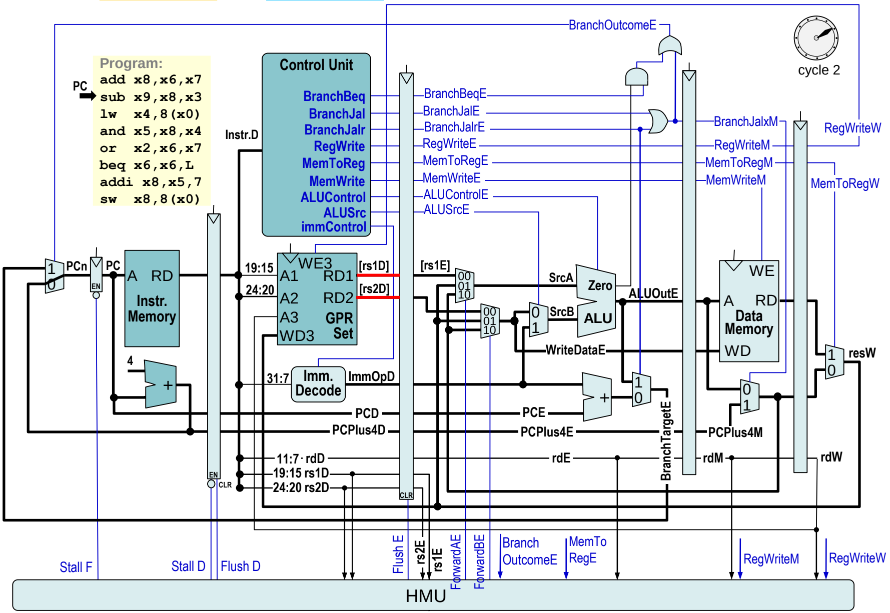


Example of Program Execution II

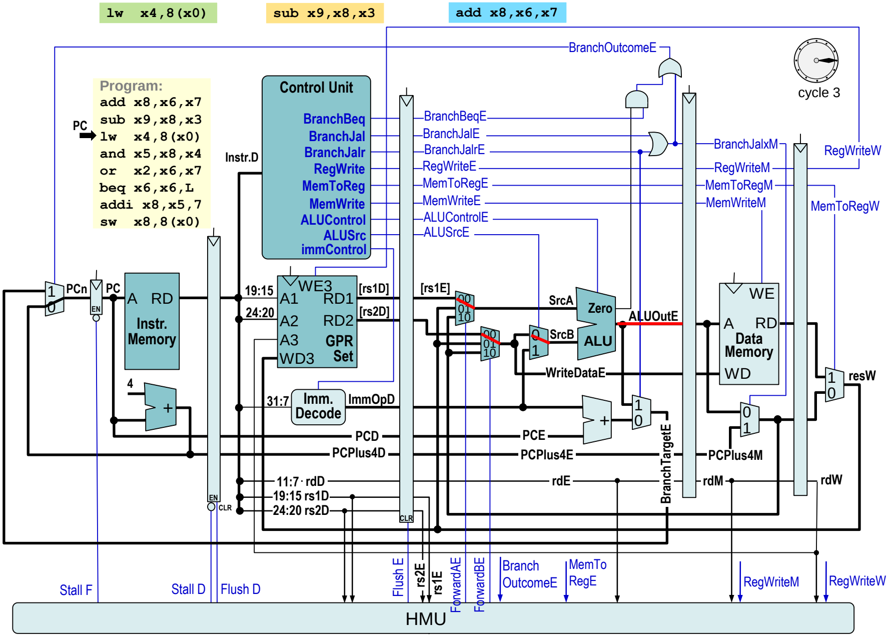
`sub x9,x8,x3`   `add x8,x6,x7`

Program:

```
add  x8,x6,x7
sub  x9,x8,x3
lw   x4,8(x0)
and  x5,x8,x4
or   x2,x6,x7
beq  x6,x6,L
addi  x8,x5,7
sw   x8,8(x0)
```



Example of Program Execution III

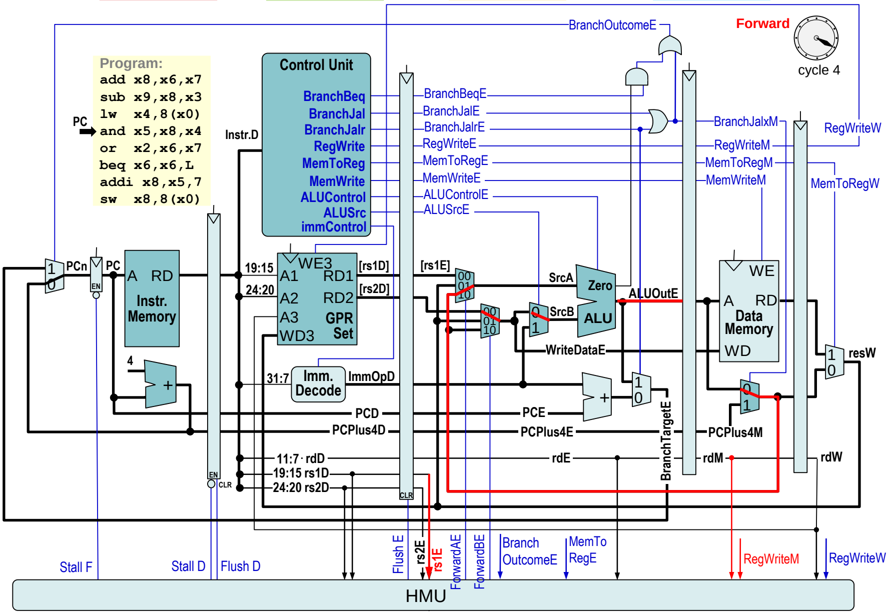


Program:

```
add  x8,x6,x7
sub  x9,x8,x3
lw   x4,8(x0)
and  x5,x8,x4
or   x2,x6,x7
beq  x6,x6,L
addi  x8,x5,7
sw   x8,8(x0)
```

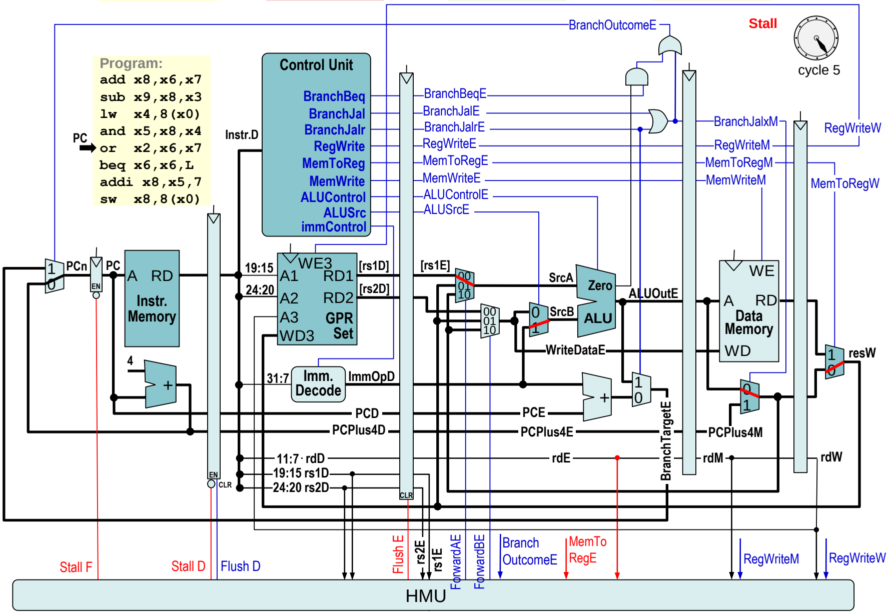
Example of Program Execution IV

`and x5,x8,x4`   `lw x4,8(x0)`   `sub x9,x8,x3`   `add x8,x6,x7`

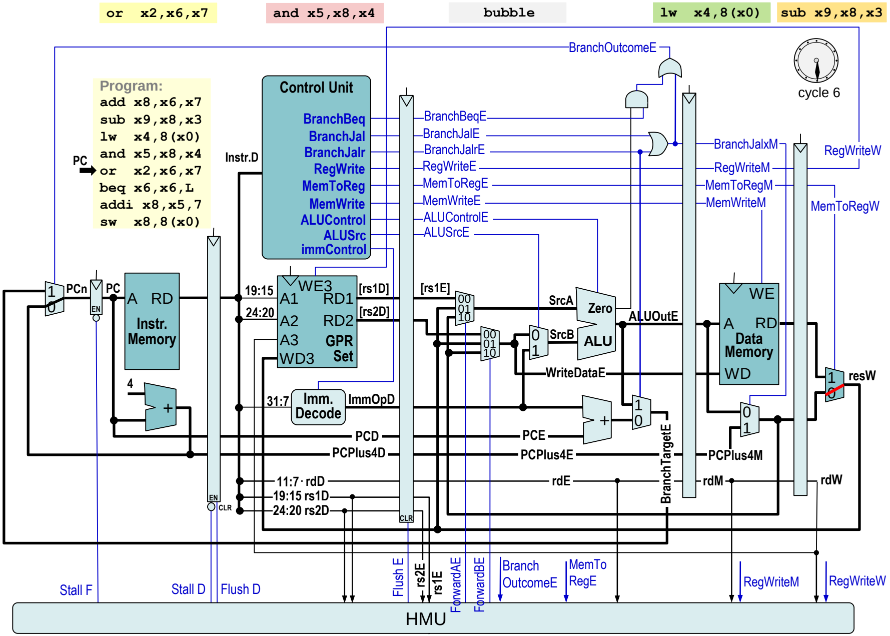


Example of Program Execution V

`or x2,x6,x7`   `and x5,x8,x4`   `lw x4,8(x0)`   `sub x9,x8,x3`   `add x8,x6,x7`

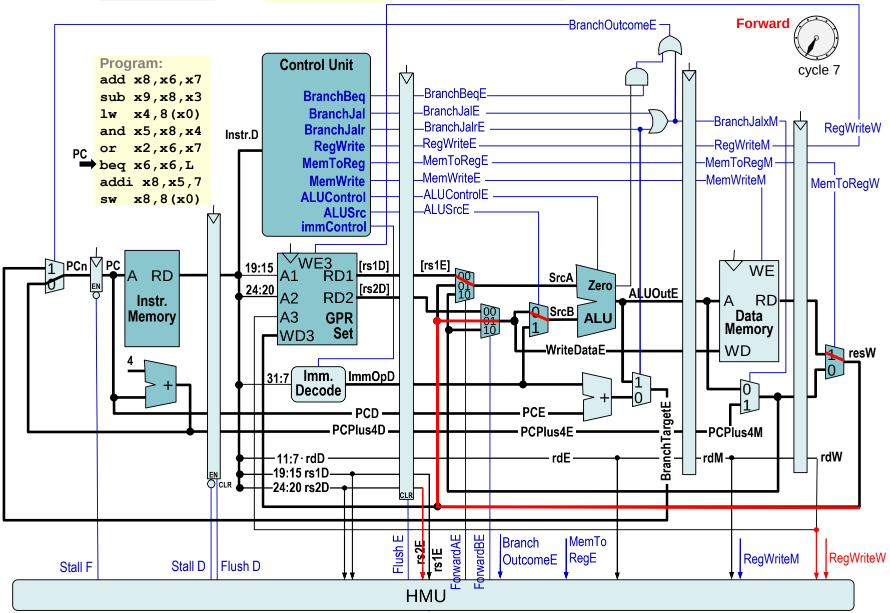


Example of Program Execution VI



Example of Program Execution VII

`beq x6,x6,L`   `or x2,x6,x7`   `and x5,x8,x4`   `bubble`   `lw x4,8(x0)`



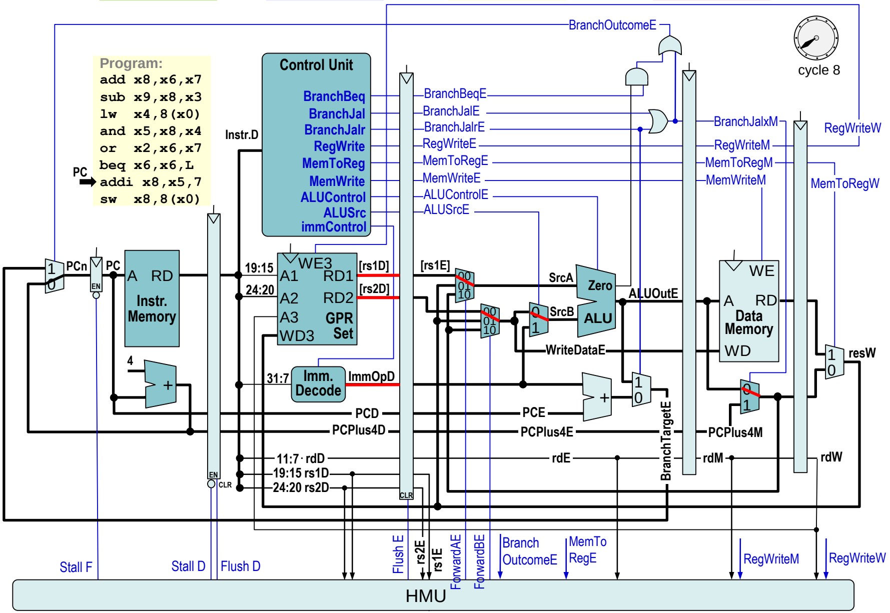
Example of Program Execution VIII

`addi x8,x5,7` `beq x6,x6,L` `or x2,x6,x7` `and x5,x8,x4` `bubble`

Program:

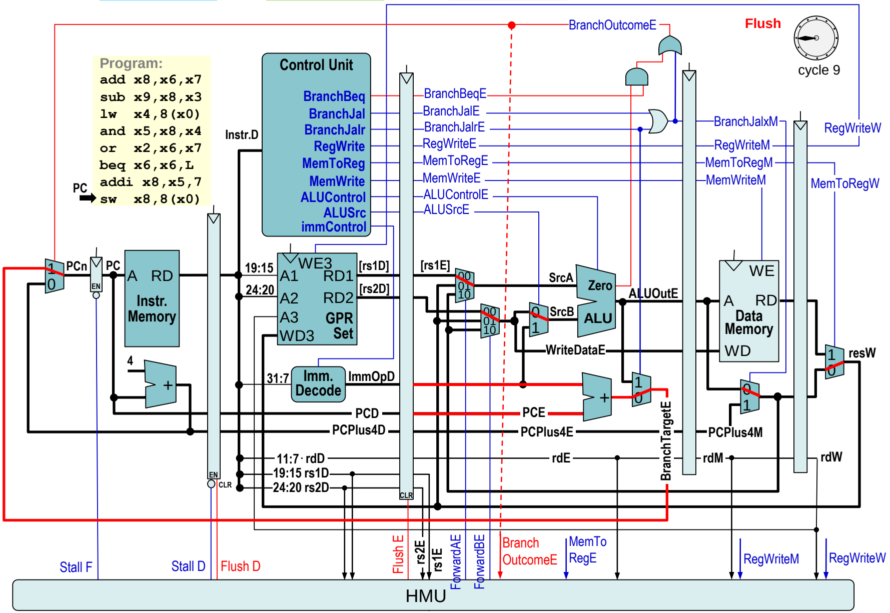
```
add  x8,x6,x7
sub  x9,x8,x3
lw   x4,8(x0)
and  x5,x8,x4
or   x2,x6,x7
beq  x6,x6,L
addi x8,x5,7
sw   x8,8(x0)
```

PC ➡ `addi x8,x5,7`



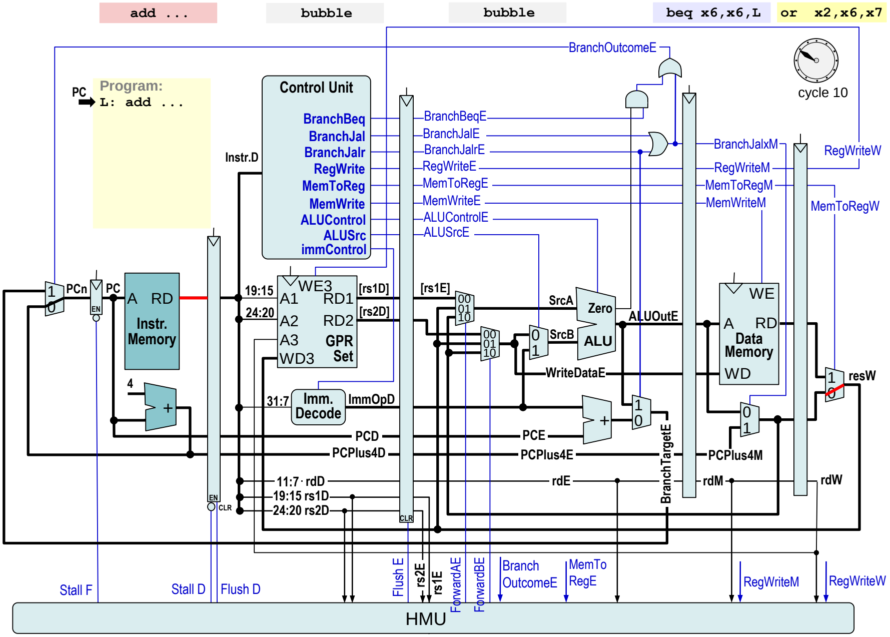
Example of Program Execution IX

`sw x8,8(x0)` `addi x8,x5,7` `beq x6,x6,L` `or x2,x6,x7` `and x5,x8,x4`



Program:
```
add  x8,x6,x7
sub  x9,x8,x3
lw   x4,8(x0)
and  x5,x8,x4
or   x2,x6,x7
beq  x6,x6,L
addi x8,x5,7
sw   x8,8(x0)
```
PC ➡ `sw x8,8(x0)`

Example of Program Execution X

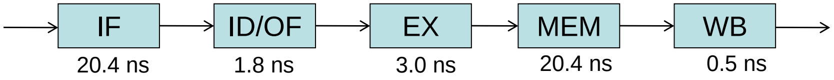


Performance of the Designed IP Microarchitecture

$$IPS = IPC / T_{CLK} = IPC * f_{CLK}$$

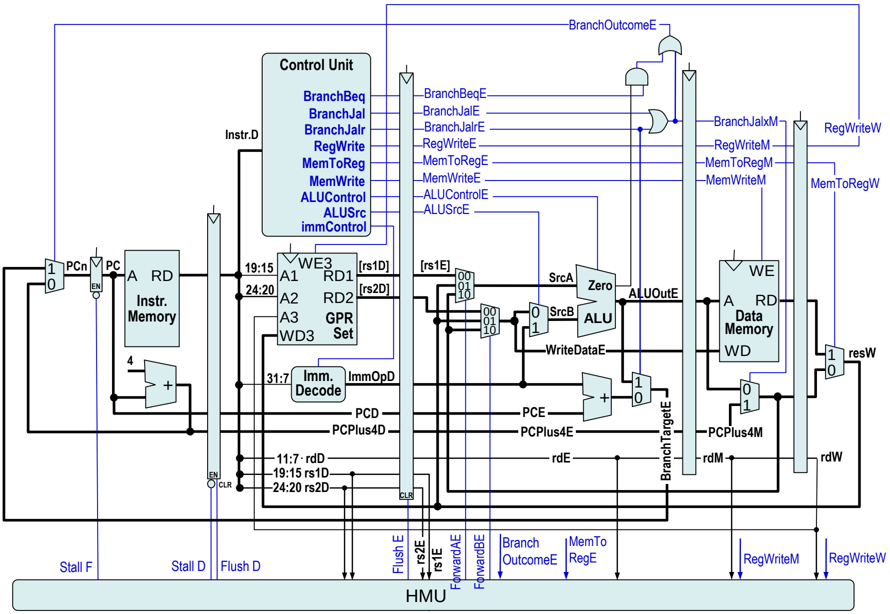
- What is the maximal acceptable frequency for the pipelined CPU if the performance parameters are those introduced in Lecture 4, Slide 31.
- The clock cycle time is determined by the slowest stage of IP.
- Which stage is the slowest one?
    - In our case, the slowest are the IF and MEM stages.
    - Their latency is given by the memory access time t_{Mem}= 20 ns plus the interstage register propagation delay $t_{RegPD} = 0.3$ ns and the interstage register setup time $t_{RegSetup} = 0.1$ ns. Thus: $T_{CLK} = 20.4$ ns => $f_{CLK} \approx 49$ MHz.
- If the IP initialization and IP stalls and bubbles are ignored, then ideal $IPC = 1$.
- $IPS = IPC * f_{CLK} = 49$ MIPS.
- **Conclusion**: Introduction of the 5-stage IP increased throughput **only!!** 2.2 times.

More Detailed Performance Analysis I

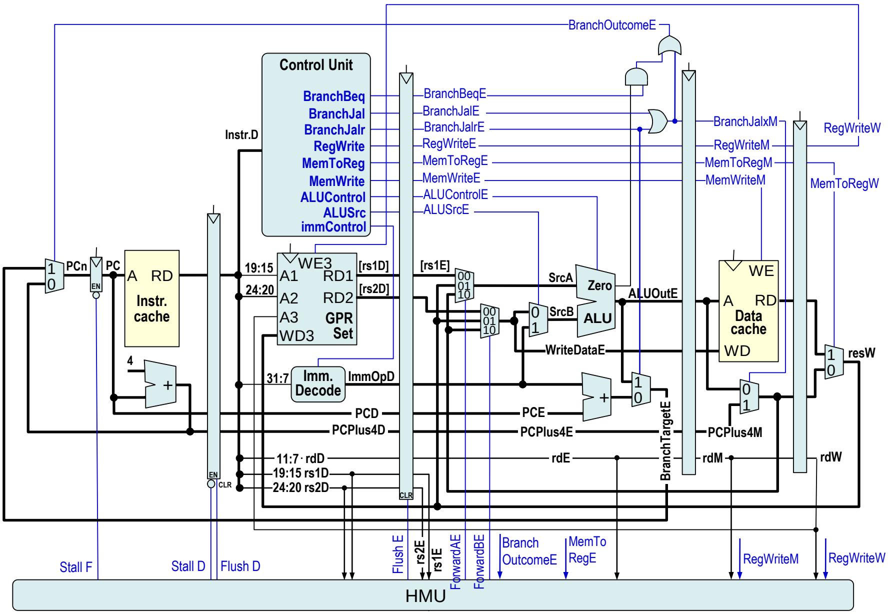


- IP is considerably **unbalanced**, i.e., the latency of individual stages significantly differ.
- As already mentioned, the slowest stages are IF and MEM ($t_{RegPD} + t_{Mem} + t_{RegSetup}$). Their latency can be substantially reduced by implementing hierarchical memory:
  - We replace both instruction and data memory with much smaller and faster **cache memory** – instruction cache memory and data cache memory. Those will be part of the processor and will communicate with external main memory (that can be shared).
  - Assume that the access latency of the cache memory is $t_{CacheMem} = 2$ ns.
  - We will devote Lecture 6 to the hierarchical memory issues.

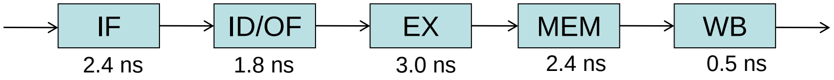
The Current Microarchitecture



Improved Design of IP Microarchitecture



More Detailed Performance Analysis II

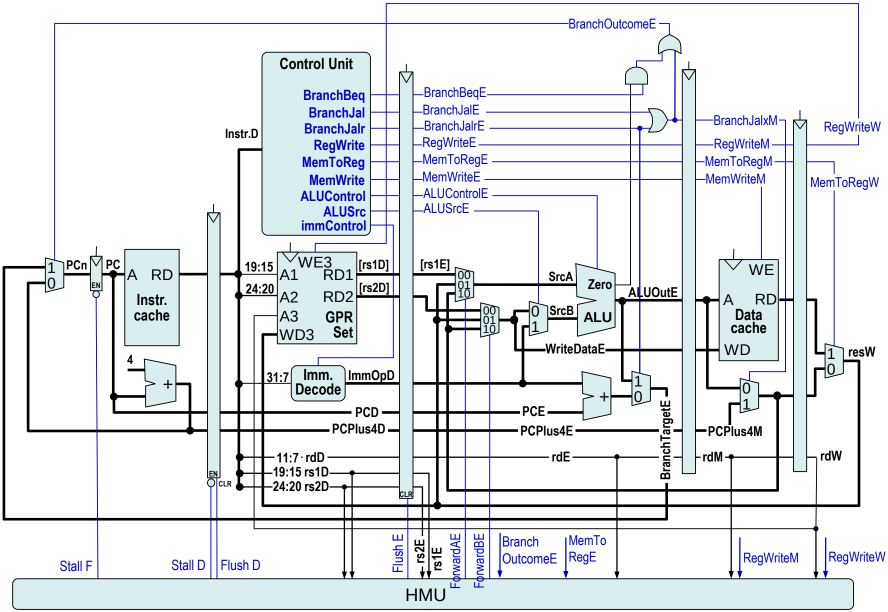


- Individual stages of the IP are better balanced.
- Now: $T_{CLK} = 3.0$ ns $=> f_{CLK} \approx 333$ MHz
- Again, if we ignore initial cold start of the IP and bubbles due to hazards, we will get:
  $IPS = IPC.f_{CLK} = 333$ MIPS.
- In case of ideal IPC, the throughput of this better balanced 5-stage IP with cache memories increased $333 / 22.7 =$ **14.7 times!** w.r.t. the single-cycle microarchitecture in Lecture 4.
- Under these conditions, the slowest stage becomes EX(!)

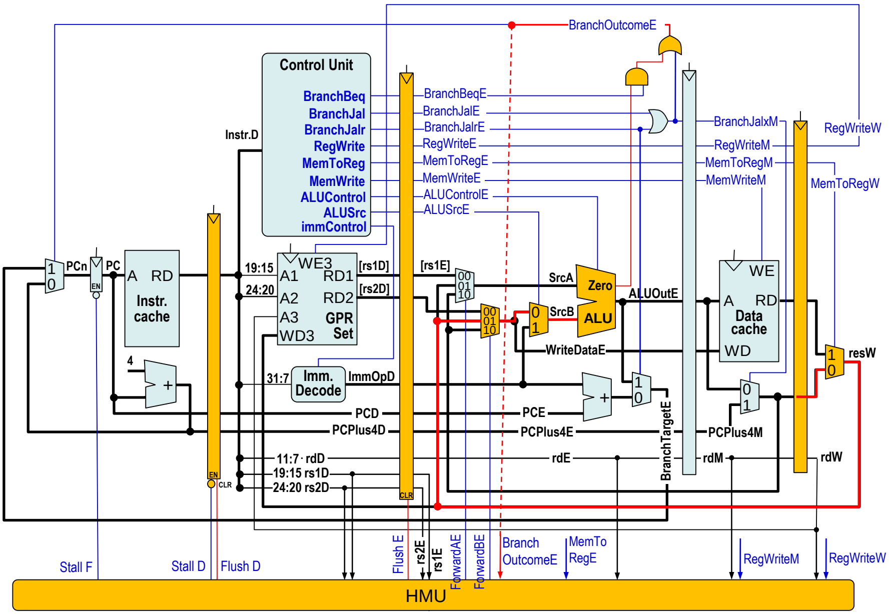
Critical Path

1. **What is a critical path?**
   - In synchronous circuits, the critical path is the path with the maximum delay in the combinational logic between two clocked registers.
2. **Why is the critical path so important?**
   - The delay of the critical path dictates the maximum clock frequency of a synchronous circuit.
3. **How to find it?**
   - Check all possible paths from all register outputs until clocked register input is reached.
   - Hint: We know that EX stage is the slowest stage of the pipeline.

Find a Critical Path



Critical Path in the Current Design – EX Stage



More Detailed Performance Analysis III

- Critical path latency is defined as:

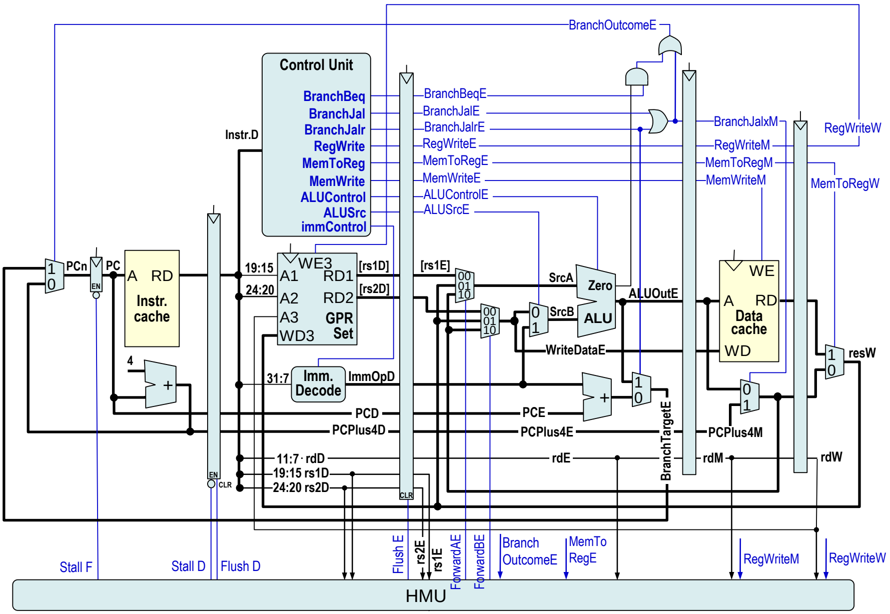
  $t_{RegPD} + 3 * t_{Mux} + t_{ALU} + t_{Gate} + t_{HMU} + t_{RegSetup} =$
  $= 0.3 + 3*0.1 + 2 + 0.1 + 0.2 + 0.1 = 3.0$ ns,

  where

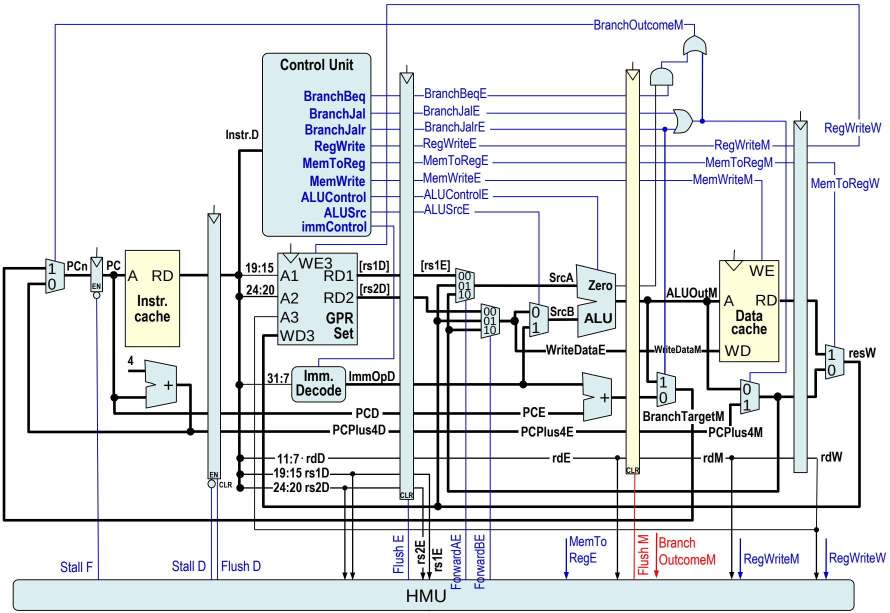
  - $t_{RegPD} = 0.3$ ns is the interstage register propagation delay,
  - $t_{Gate} = 0.1$ ns is latency of various gates generating signal `BranchOutcomeE`,
  - $t_{HMU} = 0.2$ ns is latency of the HMU.

- This latency can be reduced so that PC update with a newly computed next instruction (for instructions `beq`, `jal` and `jalr`) and reset of the interstage registers will be moved into the stage MEM. Said otherwise, we delete HMU and its combinational logic after the ALU output from the critical path.

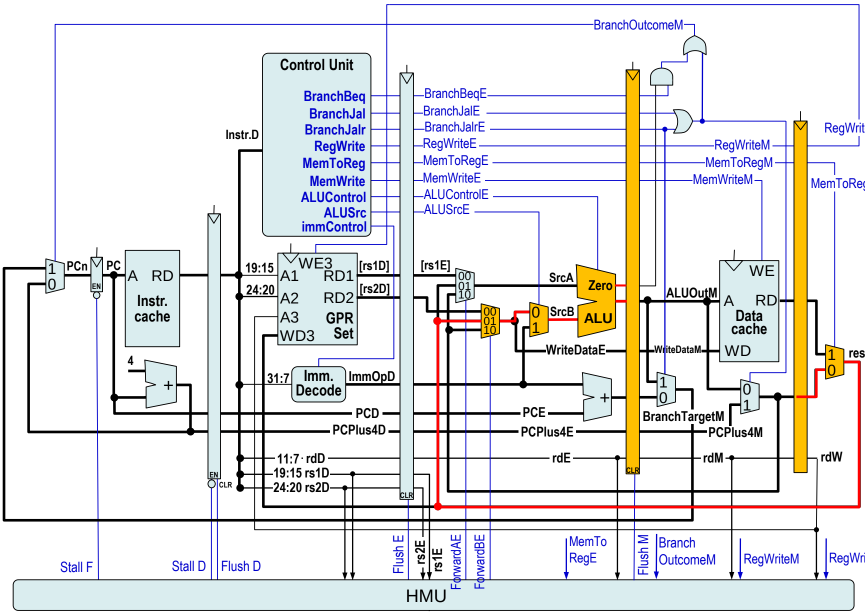
The Current Microarchitecture



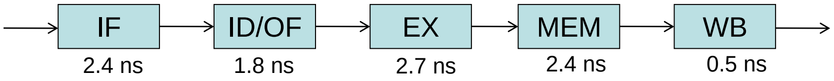
Interstage Register Shift in the IP



Critical Path after the Improvement

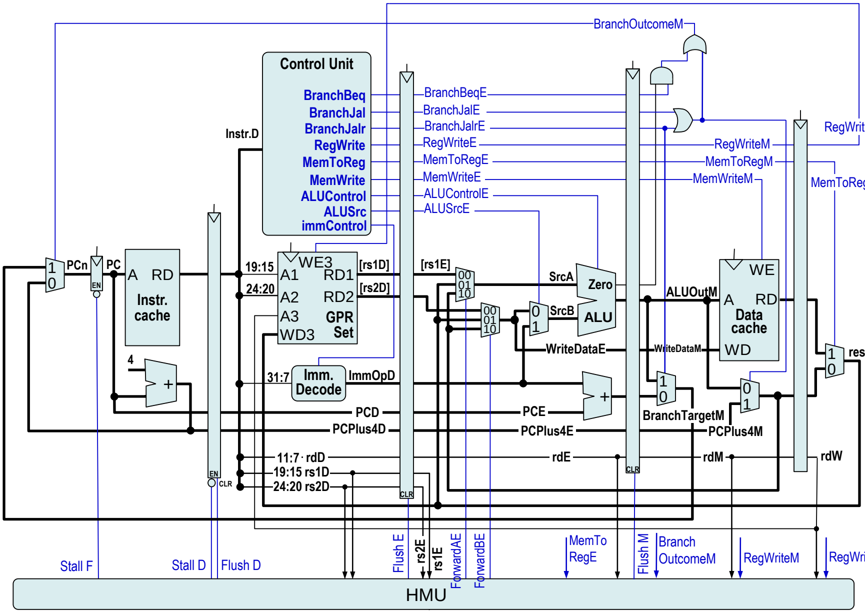


More Detailed Performance Analysis IV

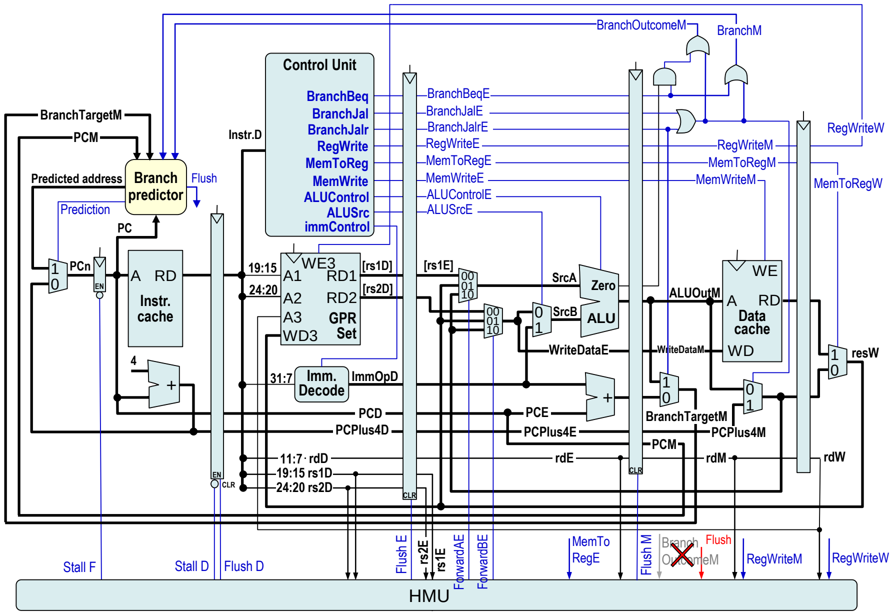


- This IP is better balanced.
- Now: $T_{CLK} = 2{,}7$ ns $\quad\Rightarrow\quad f_{CLK} \approx 370$ MHz
- Again, if we ignore initial cold start of the IP and bubbles due to hazards, we will get:
  $IPS = IPC.f_{CLK} = 370$ MIPS.
- By introducing a balanced 5-stage IP with cache memory, we improved the throughput in the ideal case 370 / 22.7 = **16.3 times!** wrt the single-cycle microarchitecture in Lecture 4.
- However at the same time, the performance is degraded due to the control hazards: Instructions `beq` (if taken), `jal`, and `jalr` now cause insertion of **3 consecutive bubbles**!
- This waste of cycles can be eliminated only by implementation of **branch prediction**. This will be discussed in detail in Lecture 13.

The Current Microarchitecture



Improved Design of an IP Microarchitecture

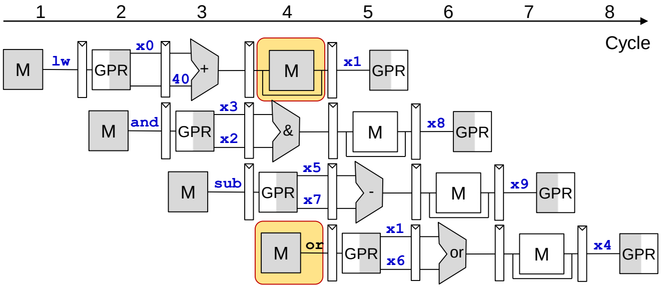


Structural hazards in pipelined CPU

- Structural hazards are collisions on shared resources (memory, GPR set, functional units).
- Our IP stages do not share resources, hence there are no structural hazards.
- In real CPUs, shared resources in IPs exist and are sources of **structural stalls**.
- The most common case is **unified memory**: simultaneous memory accesses from IF and MEM stages.
- This is why we based our picoRISC-V microarchitecture on separated instruction and data memory.

Example of Structural Hazard – Unified Memory

```
lw  x1, 40(x0)

and x8, x3, x2

sub x9, x5, x7

or x4, x1, x6
```



- The 4th instruction `or` cannot start in the 4th cycle, because instruction `lw` is accessing unified instruction and data memory.
- This hazard can be resolved by stalling the IF stage.
- When memory operands are accessed, instructions cannot be fetched, or vice versa.

Conclusions

- The instruction pipeline significantly increases CPU performance for a marginal HW cost increase.
- All contemporary CPUs use IP.
- Data dependencies between instructions may lead to hazards.
- Hazard types are:
  - data,
  - control,
  - structural.
- Resolving hazards, i.e., preserving program semantics, can be achieved by:
  (1) forwarding,
  (2) stalling combined with bubbles insertion followed by forwarding.

References

1. J. Hennessy, D. Patterson: Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 5th Edition. 2011.
2. D. Patterson, J. Hennessy: Computer Organization and Design, The HW/SW Interface. Elsevier, ISBN 978-0-12-370606-5
3. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213", Editors Andrew Waterman and Krste Asanovic, RISC-V Foundation, December 2019.
4. D. Harris, S. Harris: Digital Design and Computer Architecture, 2nd Edition. Morgan Kaufmann.