Implementation of Pipeline : *An Introduction*

Appendix A - Computer Architecture : *A Quantitative Approach* - Hennessy & Patterson

MIPS witout Pipeline: Integer Subset

- **Load/store, branch equal to zero, and the integer ALU operations are implemented.**

Instruction Fetch: IF

- **IR** $\leftarrow$ **Mem[PC]** && **NPC** $\leftarrow$ **PC + 4**.

NPC is a temporary register holding the new value of PC.

Instruction Fetch: ID

- **The instruction is decoded and**
- $\mathbf{A} \leftarrow \mathbf{Reg[rs]}$ && $\mathbf{B} \leftarrow \mathbf{Reg[rt]}$ && $\mathbf{Imm} \leftarrow \mathbf{SgExt(IR[15 \cdots 0])}$[a].

Here A, B, and Imm are temporary registers. The functional unit SgExt() sign-extends the 16-bit immediate data to 32/64-bit data

[a] What about the branch address computation?

Execute/Effective Address: EX

- **Memory Address**:
  ALUOutput $\leftarrow$ **A** $+$ **Imm**, **or**
- **Reg. - Reg. Operation**:
  ALUOutput $\leftarrow$ **A** *func* **B**, **or**
- **Reg. - Imm. Operation**:
  ALUOutput $\leftarrow$ **A** $\oplus$ **Imm**, **or**
- **Branch Operation**:
  ALUOut $\leftarrow$ **NPC** $+$ **(Imm** $<<$ **2)** && **Cond** $\leftarrow$ **(A** $==$ **0)**

Branch/Memory: MM

- **Memory Reference**:
  LMD $\leftarrow$ Mem[ALUOut] || Mem[ALUOut] $\leftarrow$ B, or
- **Branch**:
  if (Cond) PC $\leftarrow$ ALUOut

LMD (load memory data) is the data register.

Write Back: WB

- **Register-Register ALU**:
  Reg[rd] $\leftarrow$ ALUOut, or
- **Register-Immediate ALU**:
  Reg[rt] $\leftarrow$ ALUOut, or
- **Load**:
  Reg[rt] $\leftarrow$ LMD

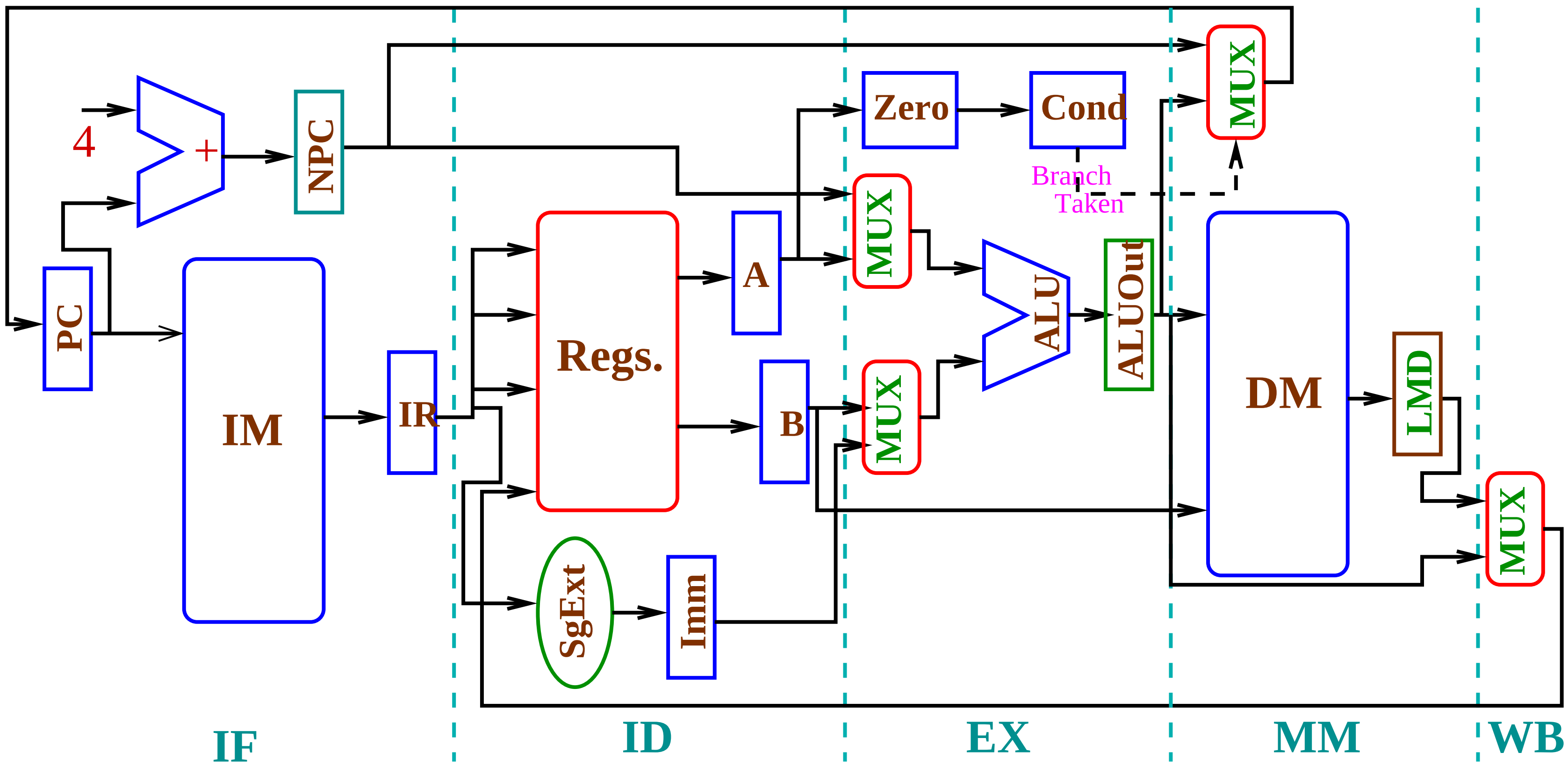

Figure 1: **A MIPS Data path (non-pipelined)**

Note · · ·

- **It is a multicycle implementation and all instructions will be completed in 4 or 5 cycles. The CPI is within 4 to 5.**
- **A simple sequential machine can be used as a controller of this processor.**
- **A microprogrammed controller can be used to control a more complex processor.**
- **There is some amount of redundancy in the data path e.g. the adder to increment the PC, separate data and instruction memory etc.**

Note · · ·

- **There can be a single cycle implementation of the processor where execution of every instruction takes one long clock cycle i.e. the CPI is 1.**
- **Any register or memory will be written at the end of the clock.**
- **There cannot be any intermediate registers.**
- **The clock cycle will be approximately five (5) times the clock of the multicycle.**
- **On an average this implementation will be very inefficient.**

A Basic Pipeline of MIPS

- The multicycle data path can be easily modified to a basic pipelined data path.
- Every pipe stage is active at every clock cycle so the operation of every stage must be completed in one cycle.
- Required values are passed from one stage to the next through the pipeline registers.

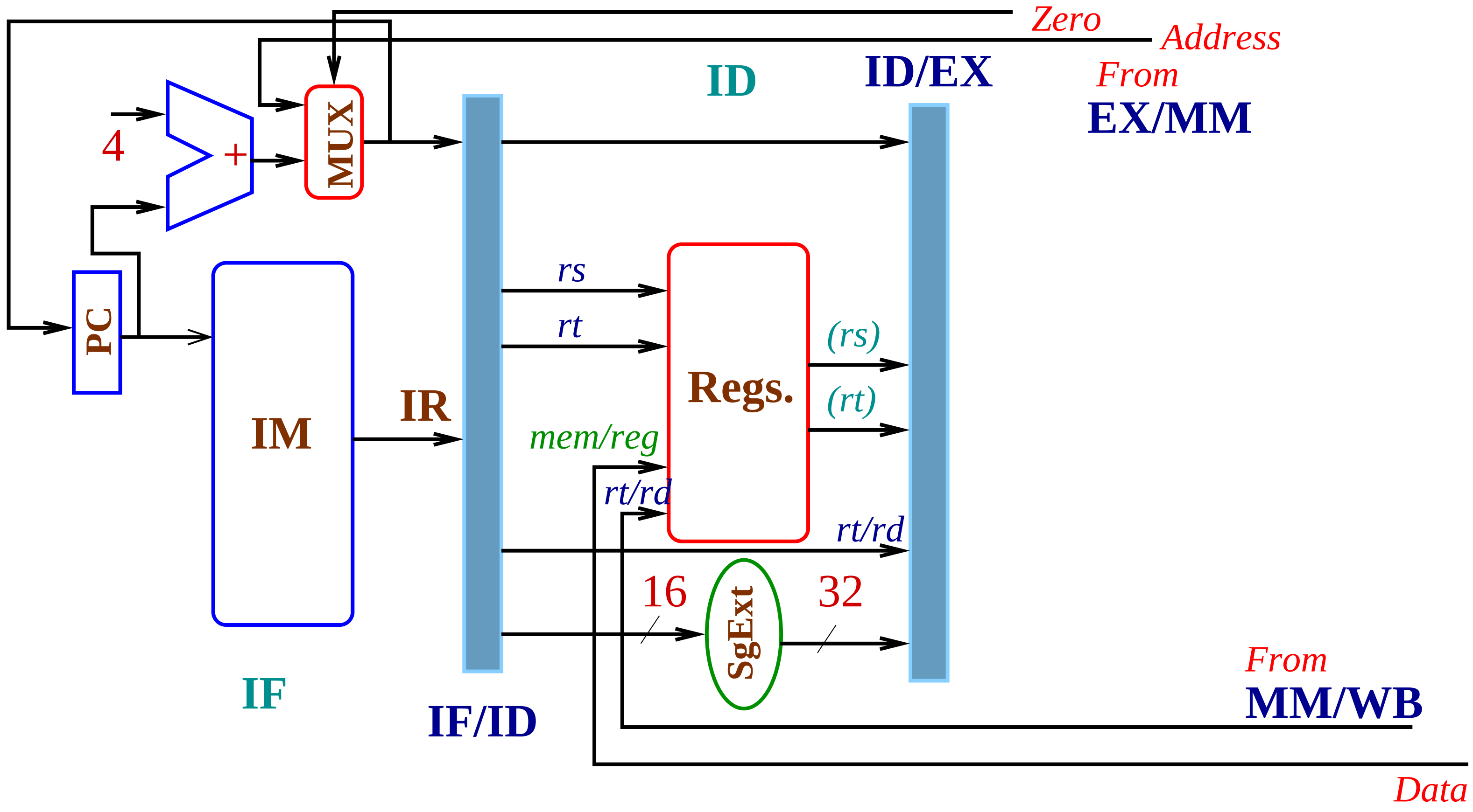

Figure 2: **A MIPS Data Path (Pipelined)**

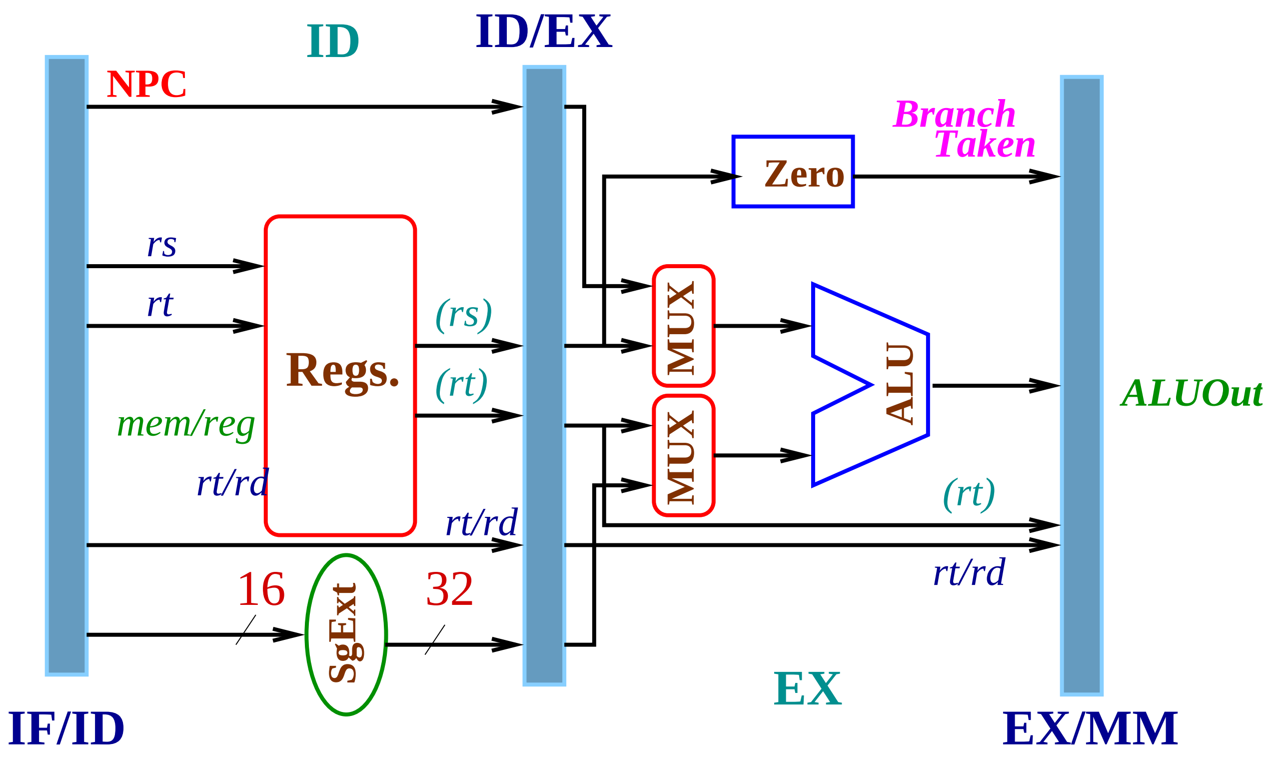

Figure 3: **A MIPS Data Path (Pipelined)**

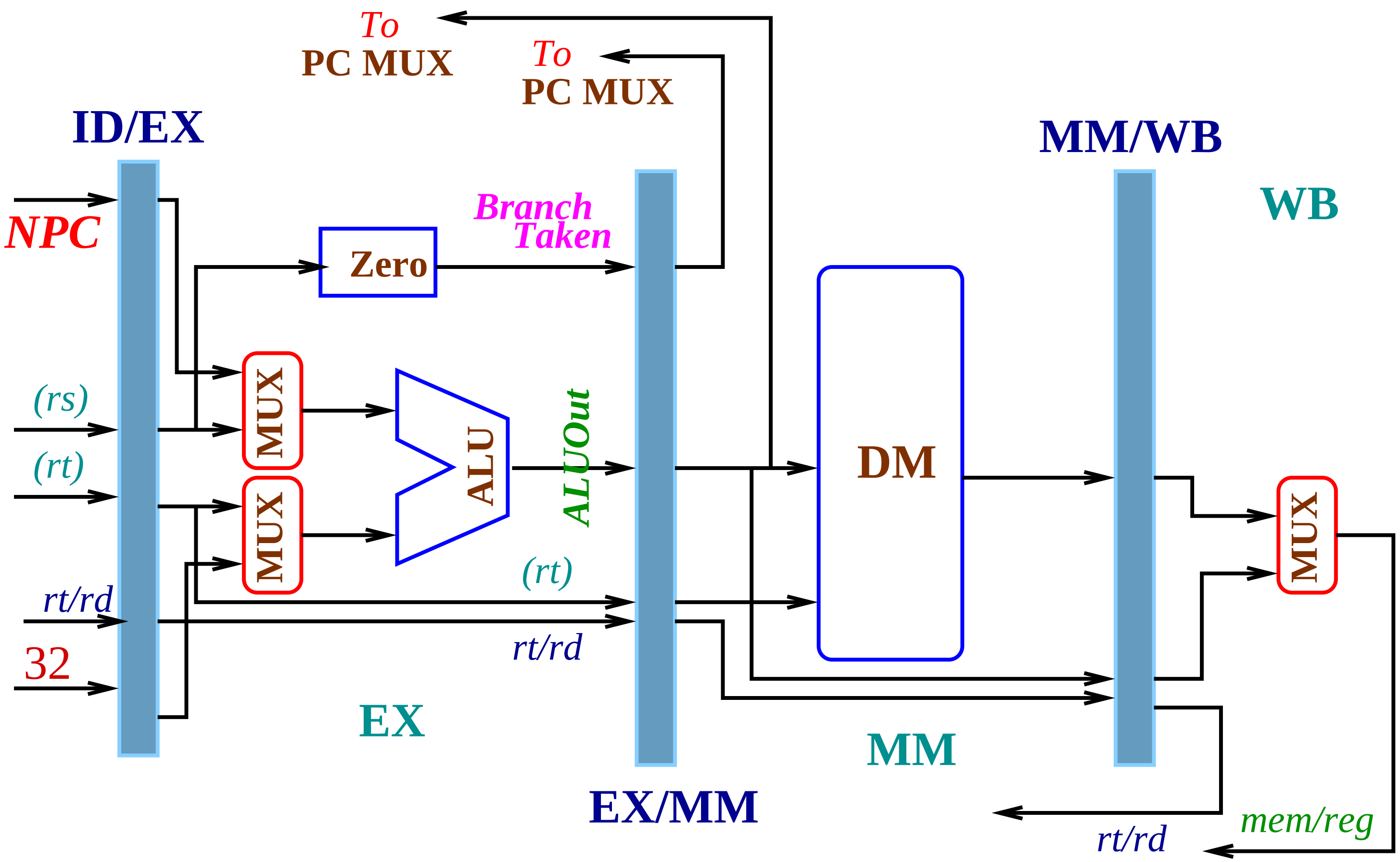

Figure 4: **A MIPS Data Path (Pipelined)**

Note ···

- The **temporary registers** of the **multicycle implementation** are **subsumed** by the **pipeline registers**.
- **Pipeline registers** are used to **carry data** and **control** that are required in the **following stages**.
- The **register is written** at the **WB stage** and the corresponding **register address** is supplied from the **MM/WB pipeline register** (**not** from the **IF/ID** which holds data for some other instruction).

Pipeline Operations: IF

```
IF/ID.IR <-- Mem[PC]
{IF/ID.NPC, PC} <-- if((EX/MM.IR[opcode] == branch) &
                        EX/MM.Cond) EX/MM.ALUOut
                    else PC + 4
```

An Example

Address	Instruction
2000	BEQZ R2, 96
2004	I1
2008	I2
2012	I3
2016	I4
...	...
2100	J1
2104	J2

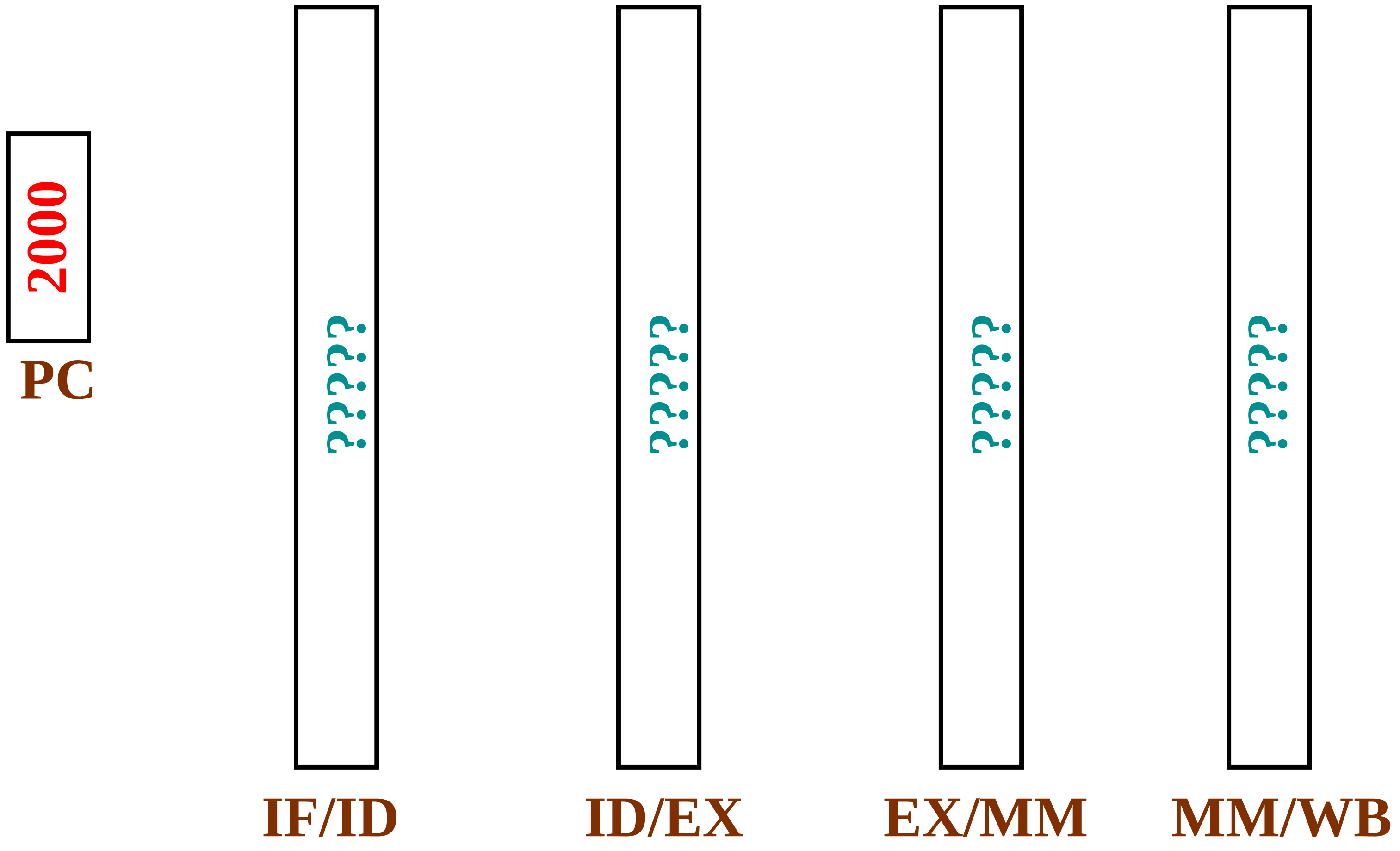

Figure 5: **At the end of t_0**

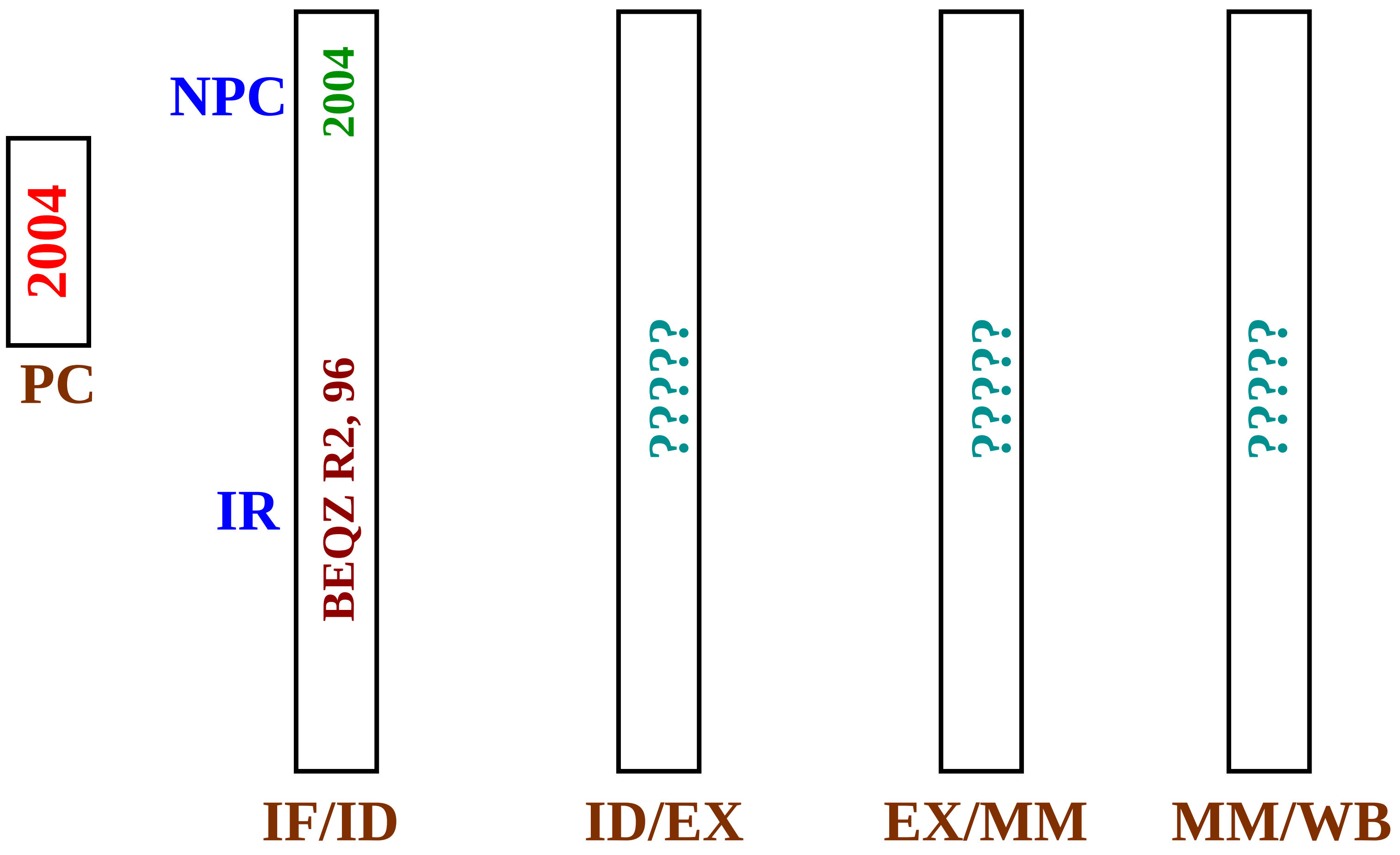

Figure 6: **At the end of $t_0 + 1$**

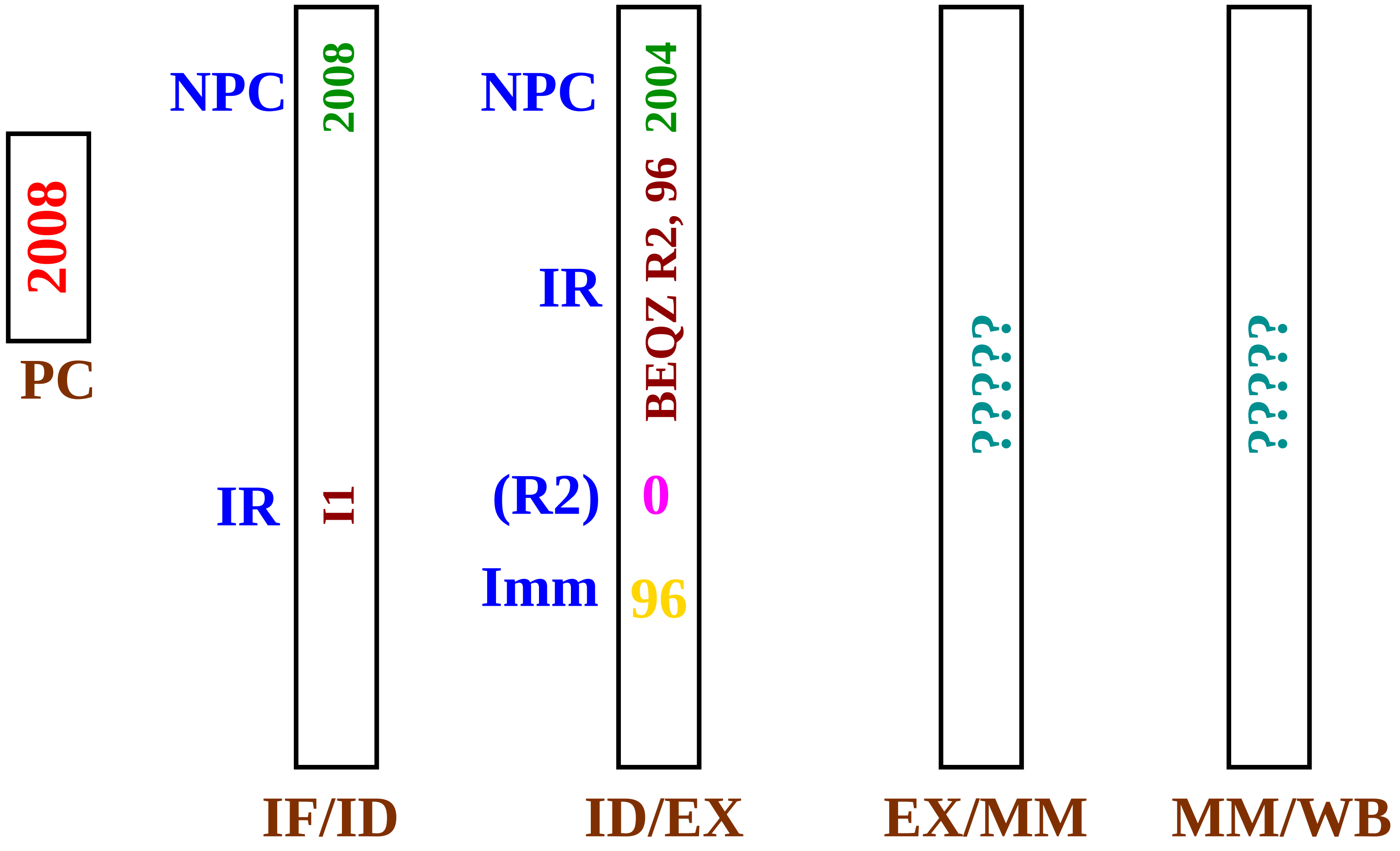

Figure 7: **At the end of $t_0 + 2$**

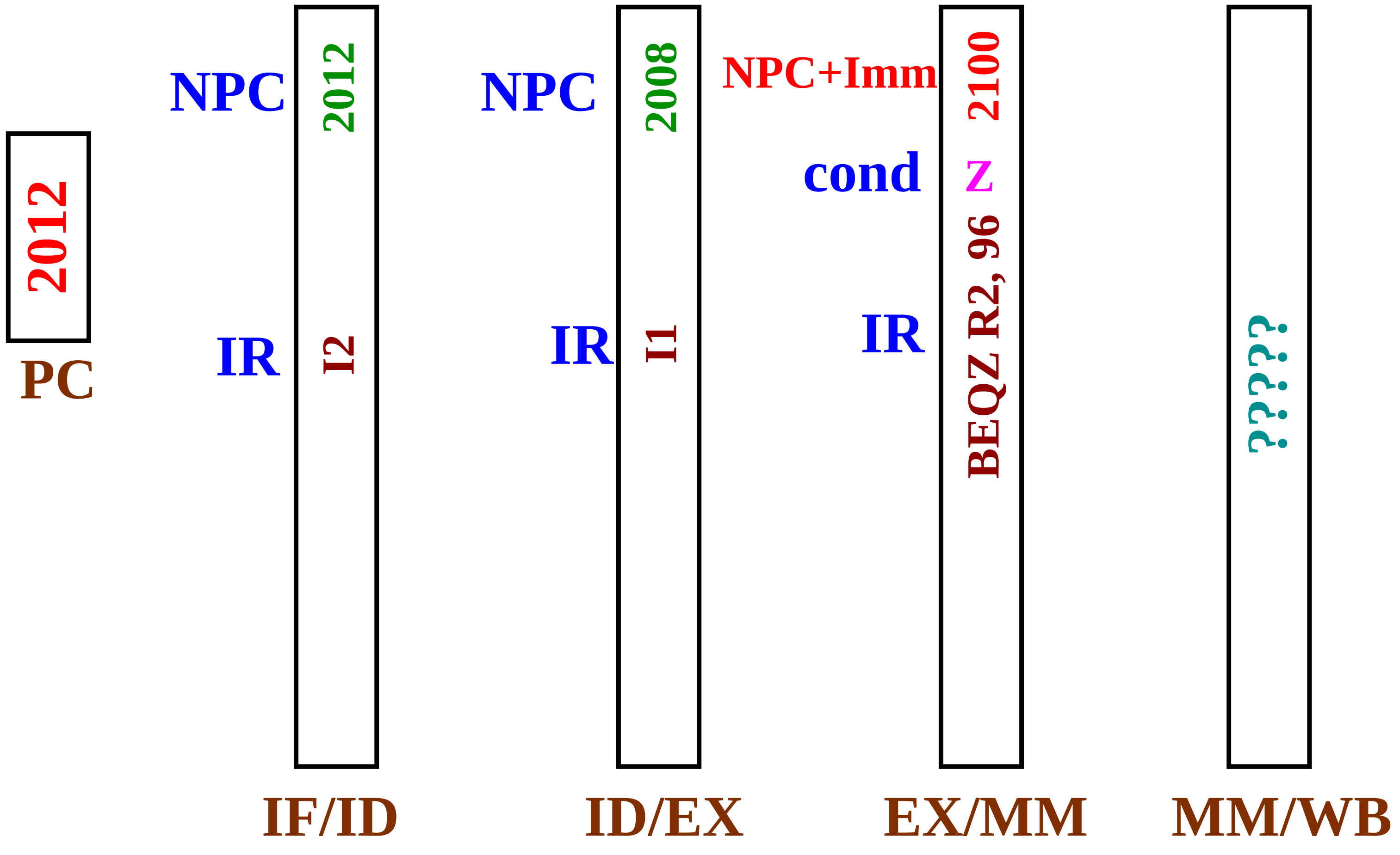

Figure 8: **At the end of** $t_0 + 3$

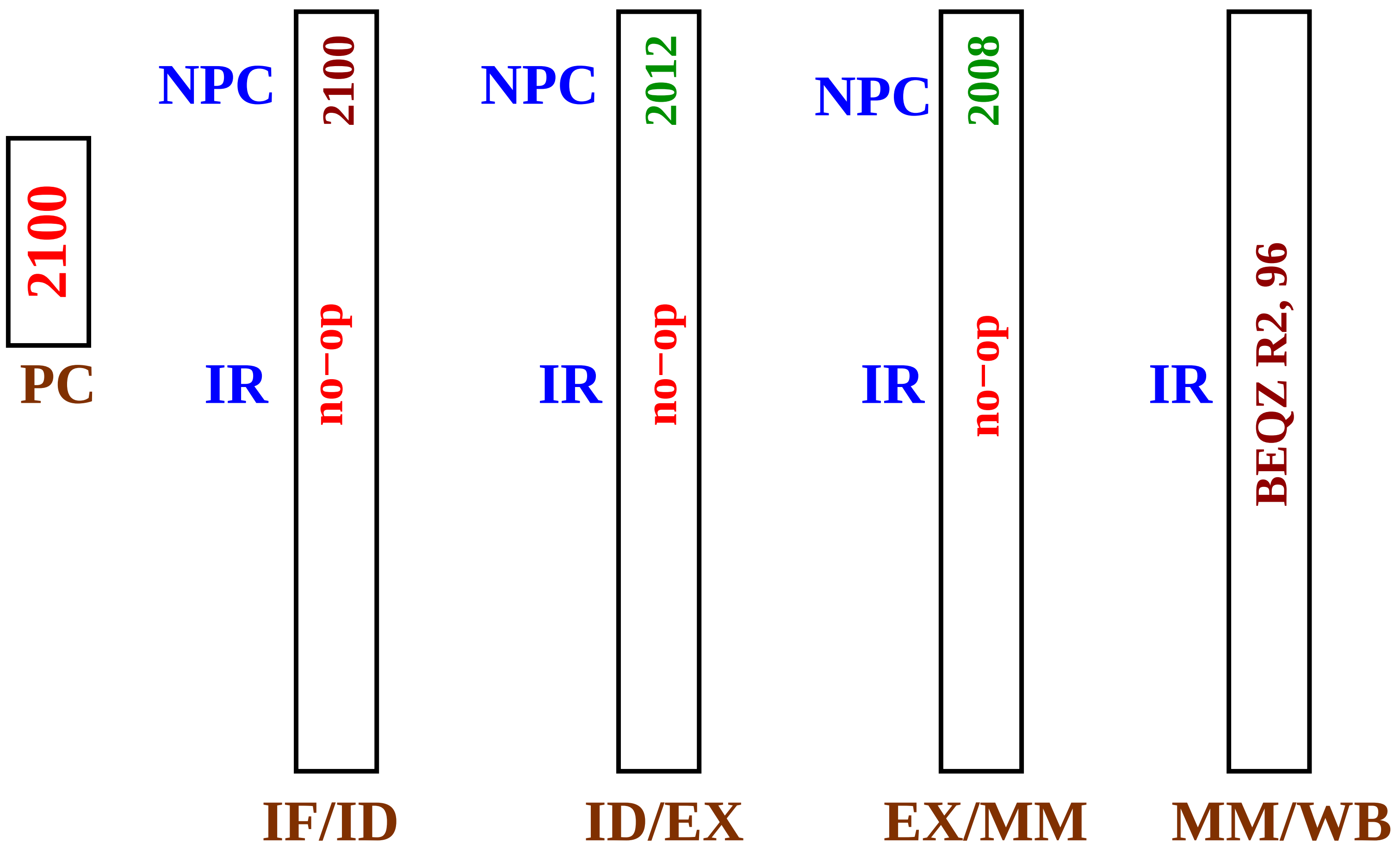

Figure 9: **At the end of $t_0 + 4$**

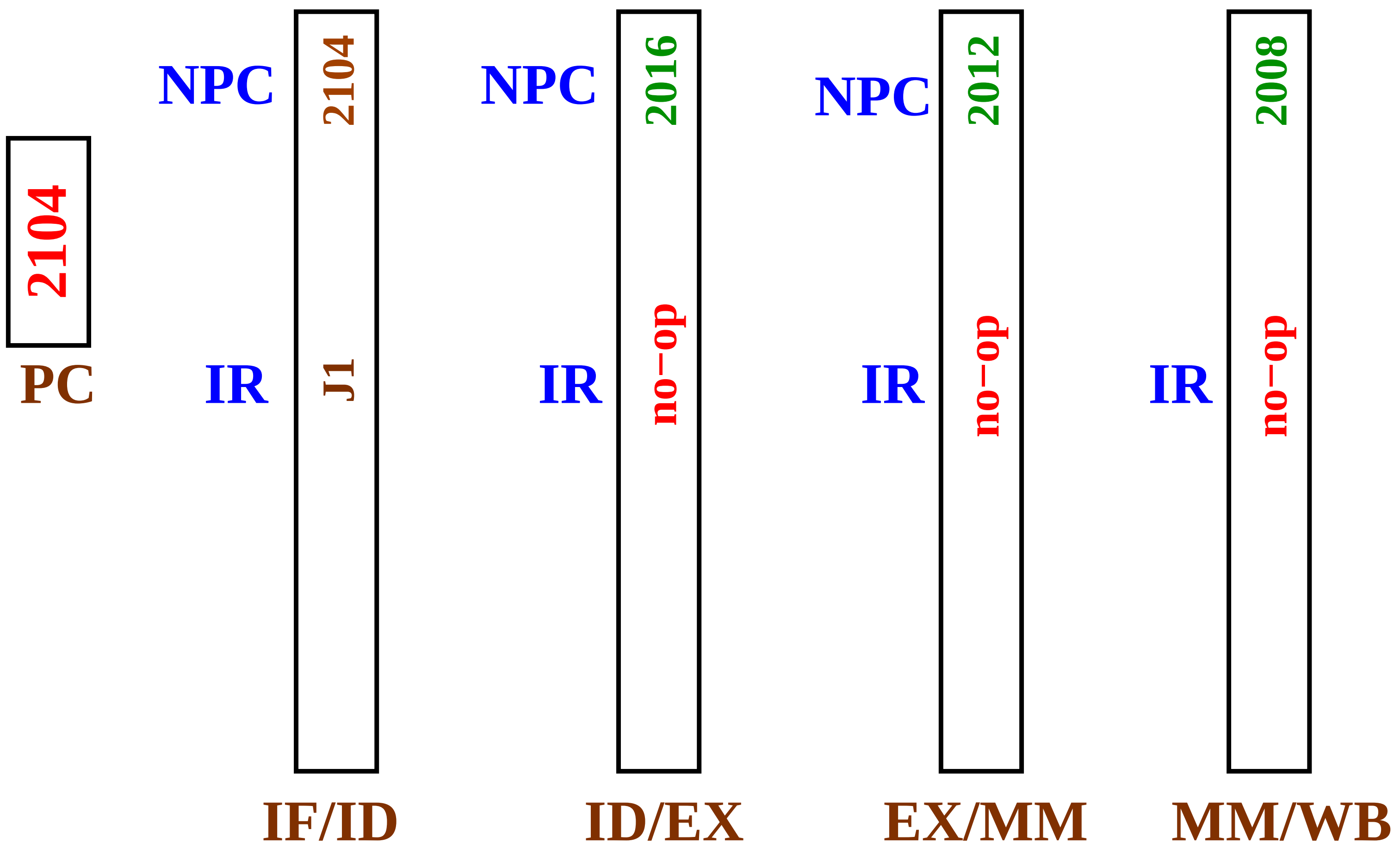

Figure 10: **At the end of $t_0 + 5$**

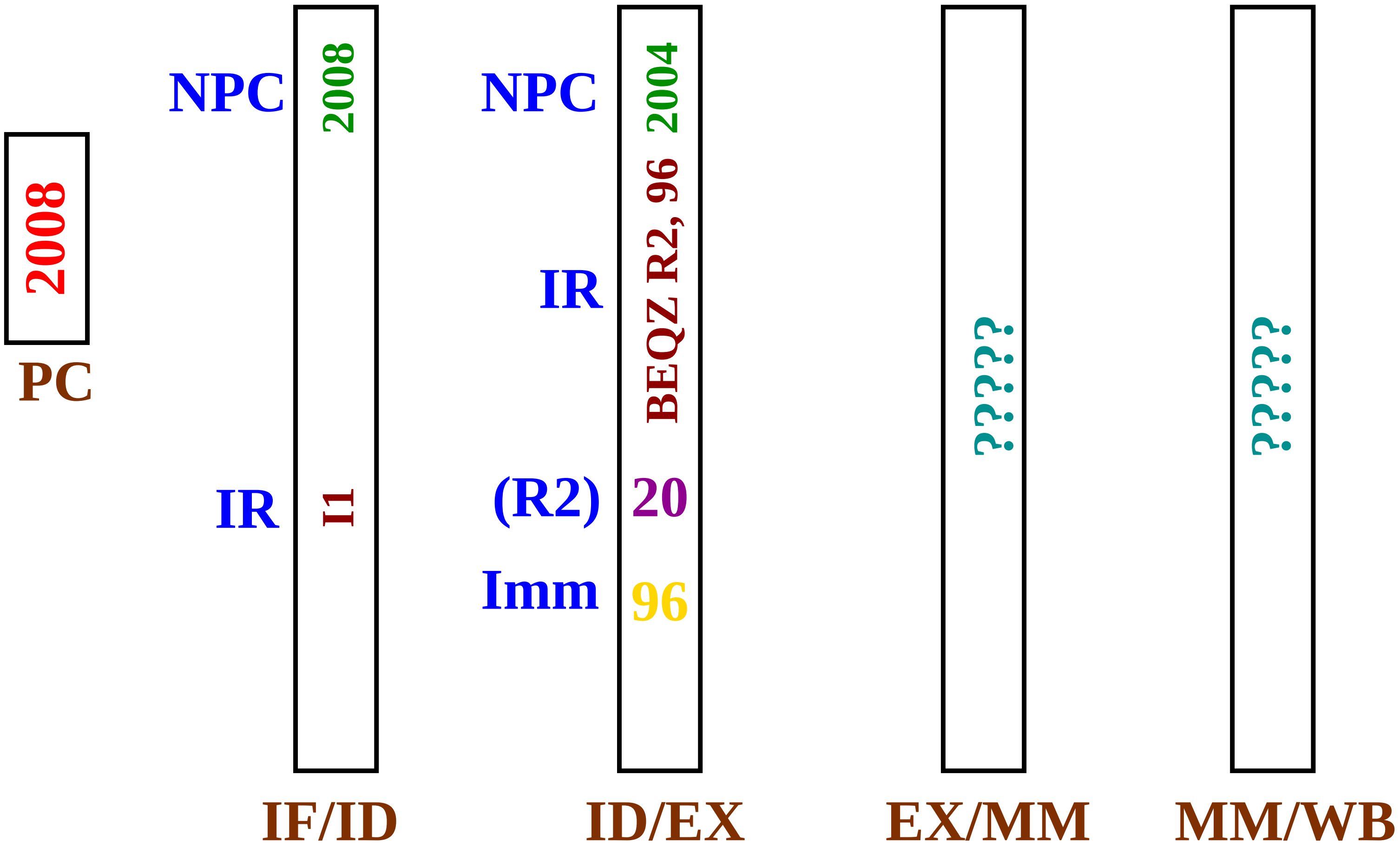

Figure 11: **At the end of $t_0 + 2$**

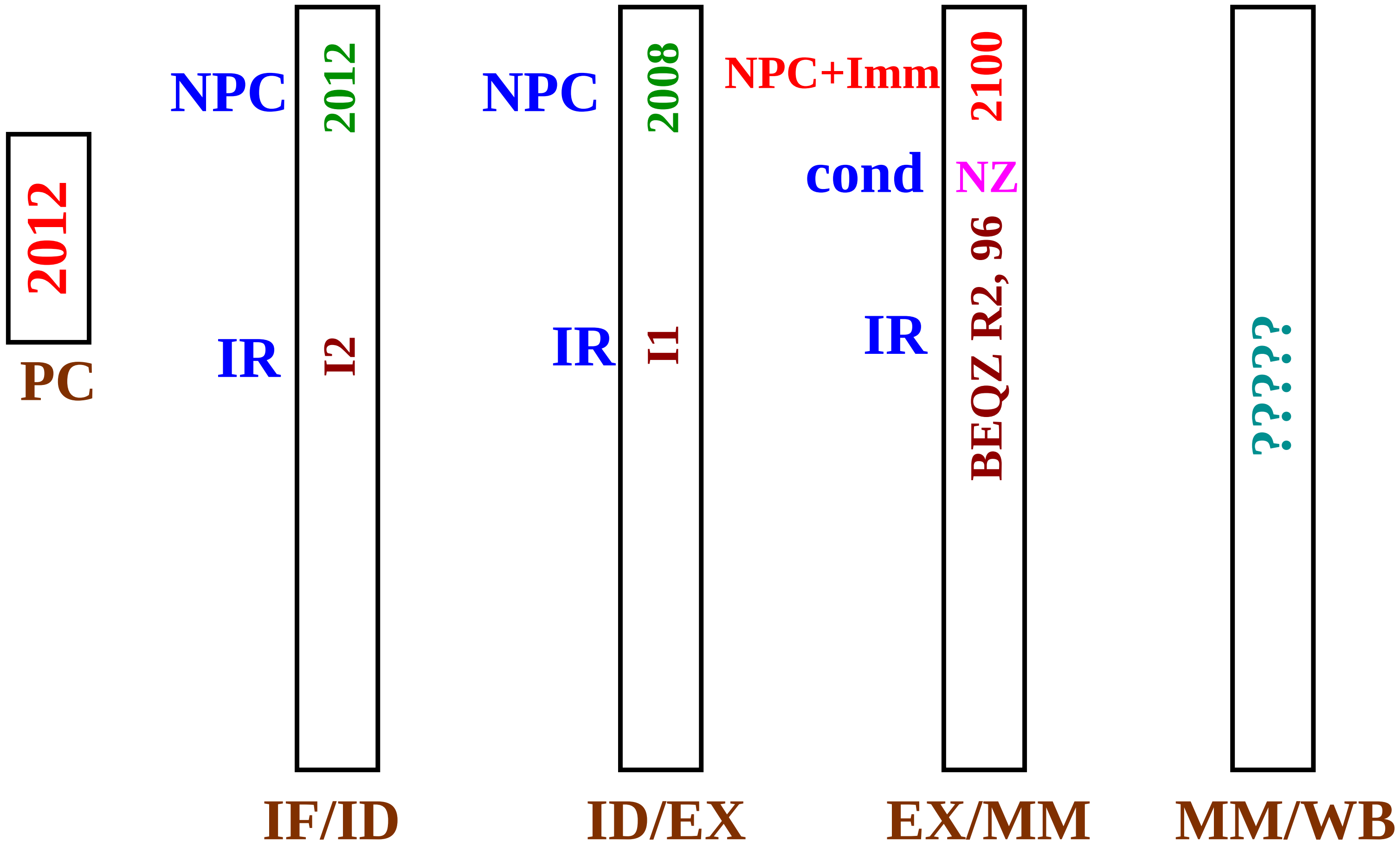

Figure 12: **At the end of $t_0 + 3$**

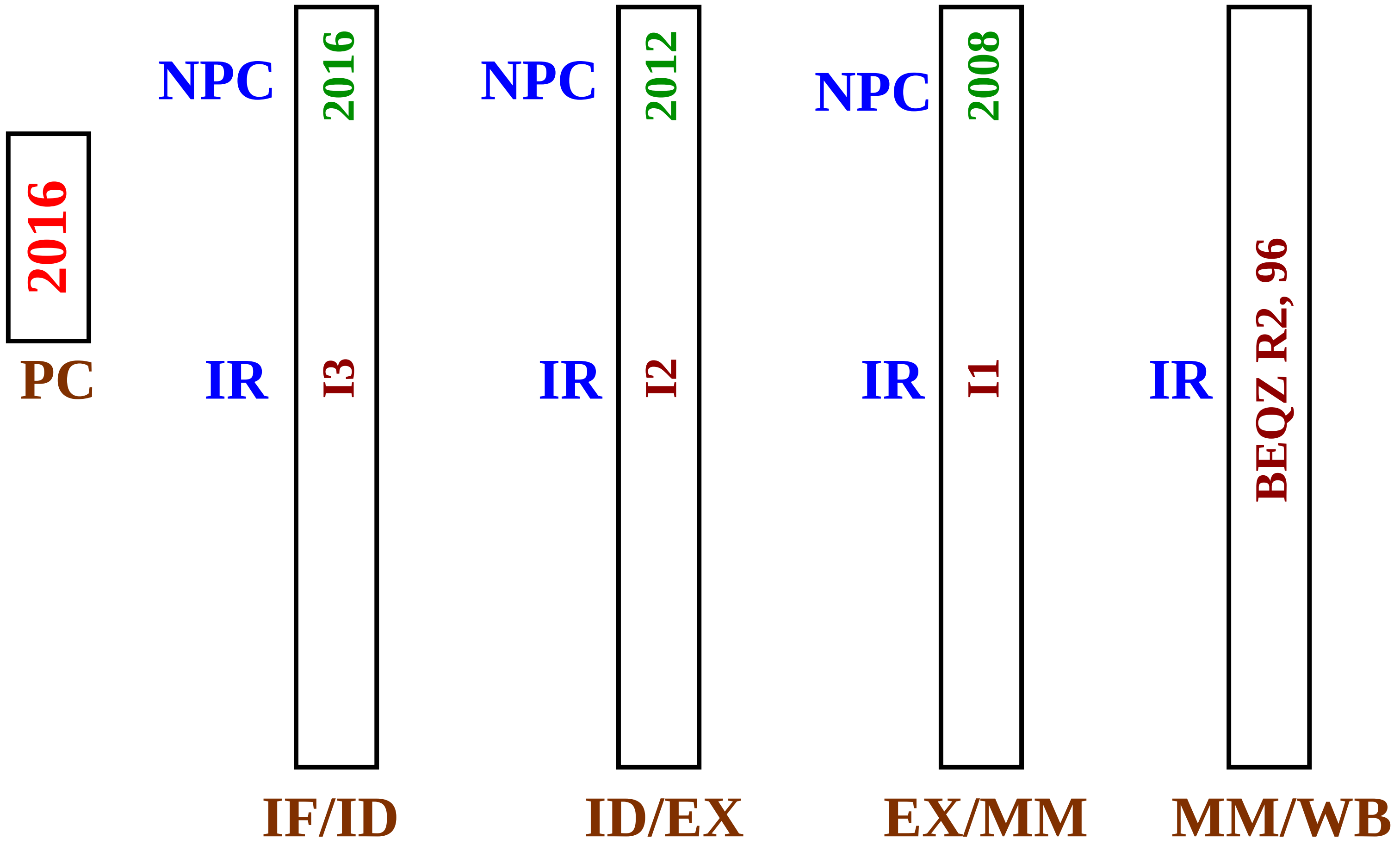

Figure 13: **At the end of** $t_0 + 4$

Pipeline Operations: ID

```
ID/EX.A <-- Reg[IF/ID.IR[rs]]
ID/EX.B <-- Reg[IF/ID.IR[rt]]
ID/EX.NPC <-- IF/ID.NPC
ID/EX.IR <-- IF/ID.IR
ID/EX.Imm <-- SgExt(IF/ID.IR[imm]])
```

Pipeline Operations: EX(ALU)

```
EX/MM.IR <-- ID/EX.IR
{
 EX/MM.ALUOut <-- ID/EX.A func ID/EX.B or
 EX/MM.ALUOut <-- ID/EX.A op ID/EX.Imm
}
```

Pipeline Operations: EX(Load/Store)

```
EX/MM.IR <-- ID/EX.IR
EX/MM.ALUOut <-- ID/EX.A + ID/EX.Imm
EX/MM.B <-- ID/EX.B
```

Pipeline Operations: EX(Branch)

```
EX/MM.NPC <-- ID/EX.NPC +
              (ID/EX.Imm << 2)
EX/MM.Cond <-- ID/EX.A == 0)
```

Pipeline Operations: MM(ALU)

```
MM/WB.IR <-- EX/MM.IR
MM/WB.ALUOut <-- EX/MM.ALUOut
```

Pipeline Operations: MM(Load/Store)

```
MM/WB.IR <-- EX/MM.IR
{
 MM/WB.LMD <-- Mem[EX/MM.ALUOut]  or
 Mem[EX/MM.ALUOut] <-- EX/MM.B
}
```

Pipeline Operations: WB(ALU)

```
{
 Reg[MM/WB.IR[rd]] <-- Mem/WB.ALUOut or
 Reg[MM/WB.IR[rt]] <-- Mem/WB.ALUOut
}
```

Pipeline Operations: WB(Load)

```
Reg[MM/WB.IR[rt]] <-- MM/WB.LMD
```

Note · · ·

- **The IF and ID stages do not depend on the current instruction as the instruction type is not yet known.**
- **The PC load in IF is controlled by the conition of Ex/MM pipeline register.**
- **The complications due to branch has not yet been considered.**
- **Branch decision taken so late will cause three clock cycle delay.**

MUX Control

- The PC/NPC MUX is controlled by the EX/MM.Cond - whether the branch is taken.
- The top ALU MUX is controlled by the ID/EX.IR - whether a branch instruction.
- The bottom ALU MUX is also controlled by ID/EX.IR - whether register-register operation.
- The WB stage MUX is controlled by MM/WB.IR - whether load or ALU operation.
- Hidden MUX for rt/rd - whether register-register or immediate.

Control the MIPS Pipeline

- **Instruction Issued: an instruction is said to be issued when it moves from ID to EX.**
- **Data hazard in MIPS pipeline can be detected at ID stage - the instruction can be stalled or appropriate forwarding can be made if possible.**

Control the MIPS Pipeline

- **Early detection of interlock among instructions in the pipeline reduces the hardware complexity as no instruction that has already changed the state e.g. decrement register, is necessary to suspended.**

Control the MIPS Pipeline

- **An alternative possibility is to detect the hazard or forwarding requirement at the begining of the clock cycle (EX or MM) where the operand will be used. We consider two examples -**
- **Interlocking for read-after-write (RAW) in case of load operand. It is detected at ID.**
- **Implementation of forwarding path to the ALU during EX cycle.**

Different Situations: Load (L)

No Dependence	
LD	R1, 45(R2)
DADD	R3, R4, R5
DSUB	R6, R7, R8
OR	R9, R10, R11

Different Situations: Load (L)

	Stall
LD	R1, 45(R2)
DADD	R3, R1, R5
DSUB	R6, R7, R8
OR	R9, R10, R11

The use of R1 is detected by a comparator and DADD is stalled.

Different Situations: Load (L)

Stall	
LD	R1, 45(R2)
DADD	R3, R4, R5
DSUB	R6, R1, R8
OR	R9, R10, R11

The use of R1 is detected by a comparator and data is forwarded to ALU when DSUB is in EX.

Different Situations: Load (L)

Stall	
LD	R1, 45(R2)
DADD	R3, R4, R5
DSUB	R6, R7, R8
OR	R9, R1, R11

No action required.

Load RAW Interlock

- **The load instruction is in the EX stage.**
- **The instruction that uses loaded data is in ID stage.**
- **Three comparisons are required.**

Operand Test for Hazard and Forwarding

ID/EX.IR[0···5] is load

- IF/ID.IR[0···5] is register-register: ID/EX.IR[rt] == IF/ID.IR[rs] and ID/EX.IR[rt] == IF/ID.IR[rt].
- IF/ID.IR[0···5] is load, store, ALU Imm., branch: ID/EX.IR[rt] == IF/ID.IR[rs].

After Detection of Hazard

- A pipeline stall is inserted. Instructions from ID and IF stages are prevented to advance.
- The content of ID/EX.IR is changed to no-op.
- The IF/ID.IR is reloaded with its old value.
- PC value is not to be incremented.

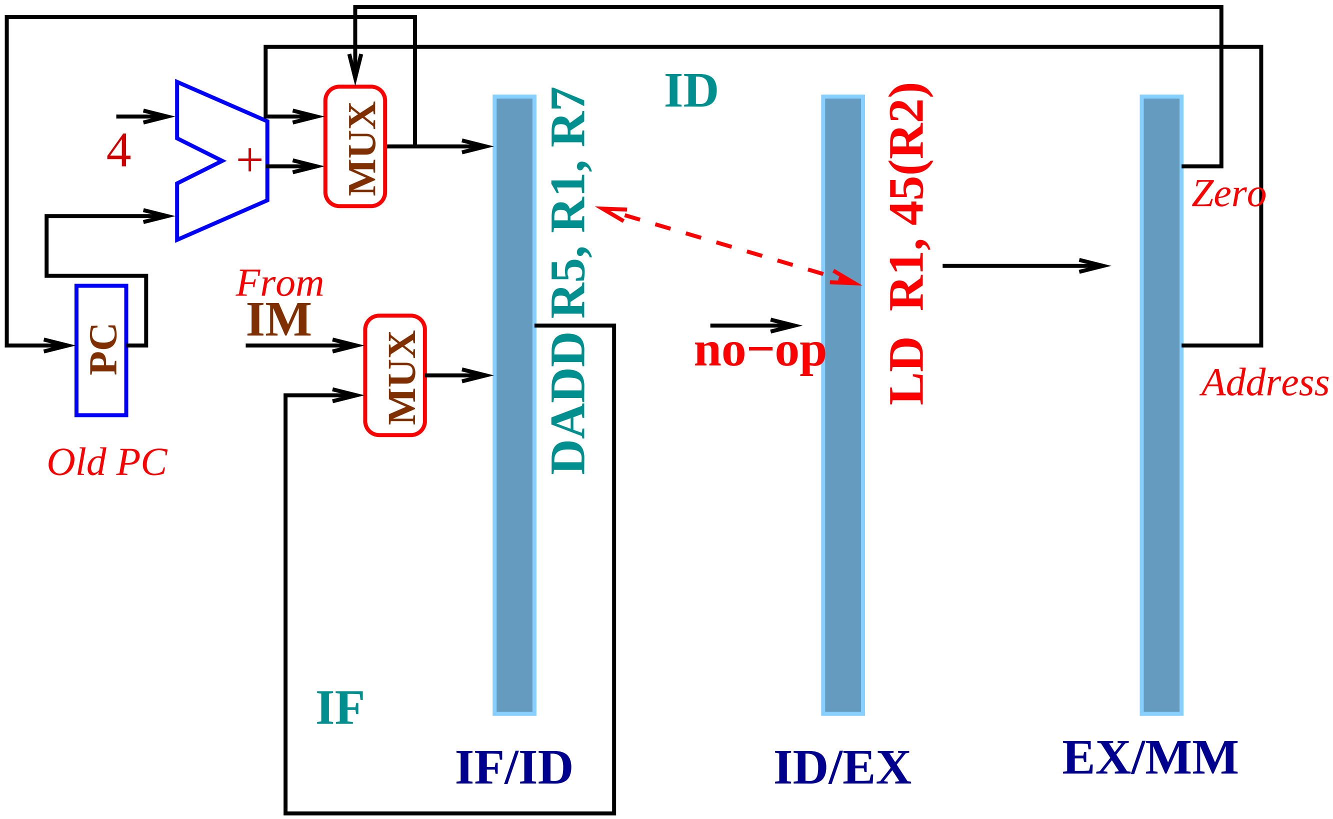

Figure 14: **A Hazard in MIPS Data Path**

Data Forwarding

- **Pipeline registers contain data to be forwarded and the register addresses for the source and the destination registers.**
- **Logically any forwarding takes place either from the ALU or from the LMD (data-memory register).**
- **We need to compare the destination registers of EX/MM and MM/WB stages with the source registers in the ID/EX.**
- **There are ten (10) possible cases.**

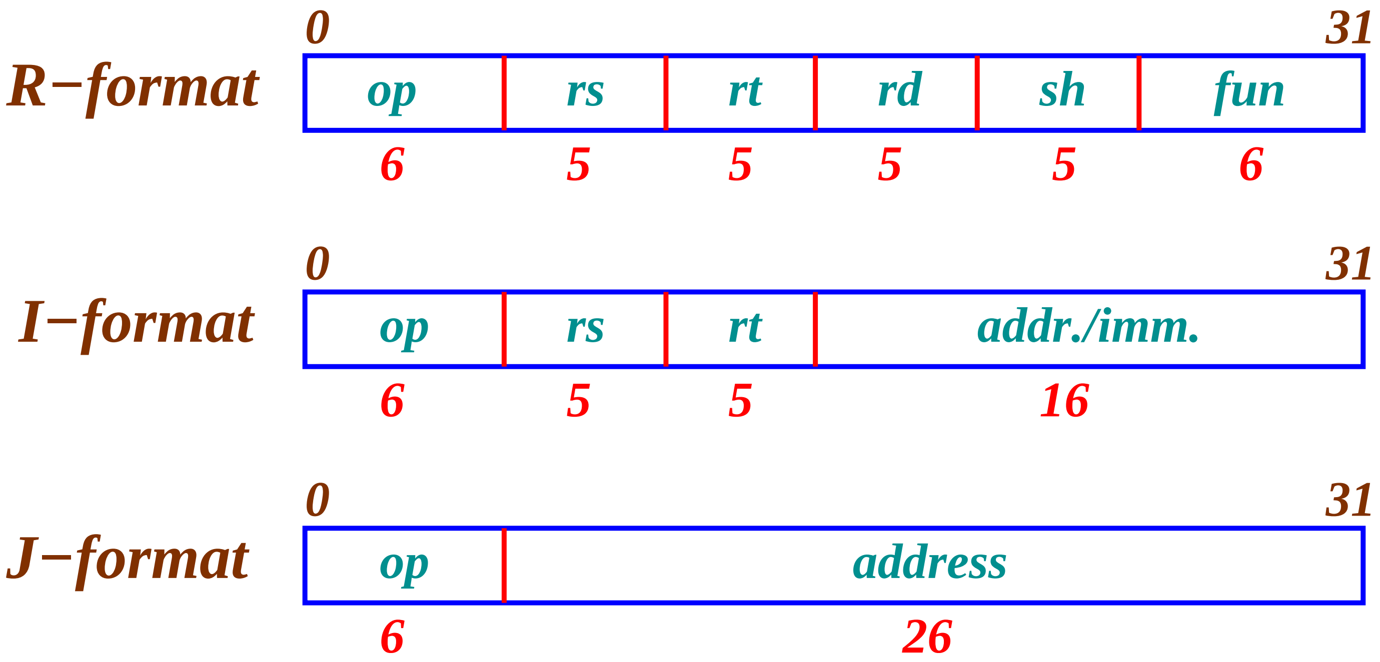

Figure 15: **MIPS Instruction Formats**

Forwarding from EX/MM

- **Data source** is a register-register ALU instruction at EX/MM: DADD R3, R2, R1
- Data to forward is in EX/MM.ALUOut.
- The **data destination** instructions are: DSUB R4, R3, R5, DADDI R4, R3, data, LD R4, off(R3), SD R4, off(R3), BEQ R4, R3, target .
- Compare EX/MM.IR[rd] == ID/EX.IR[rs].

An Example

- **Source**: **DADD R3 (rd), R2 (rs), R1 (rt)**:
  R3 $\leftarrow$ R2 + R1
- **Destination**: **LD R4 (rt), off(R3) (rs)**:
  R4 $\leftarrow$ Mem[R3 + off].
- **Sequence**:
  DADD R3, R2, R1
  LD R4, off(R3)
- **Compare**: **EX/MM.IR[rd] == ID/EX.IR[rs].**
- **The EX/MM.ALUOut is to be forwarded to the rs-MUX of the ALU.**

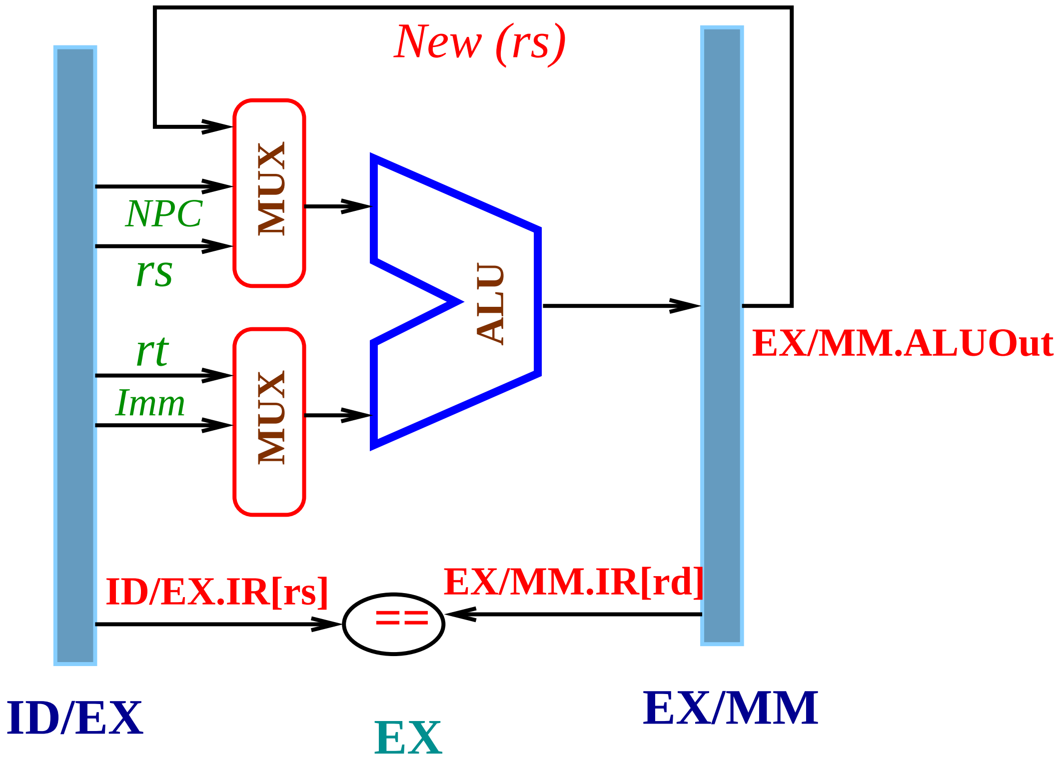

Figure 16: **Forwarding in MIPS Data Path**

Forwarding from MM/WB

- **Source**: **DADD R3 (rd), R2 (rs), R1 (rt)**
- **Destination**: **DADDI R4 (rt), R3 (rs), imm**: **R4 ← R3 + imm.**
- **Sequence**:
  DADD R3, R2, R1
  LD R5, off(R1)
  DAAI R4, R3, imm.
- **Compare**: **MM/WB.IR[rd] == ID/EX.IR[rs].**
- **The MM/WB.ALUOut is to be forwarded to the rs-MUX of the ALU.**

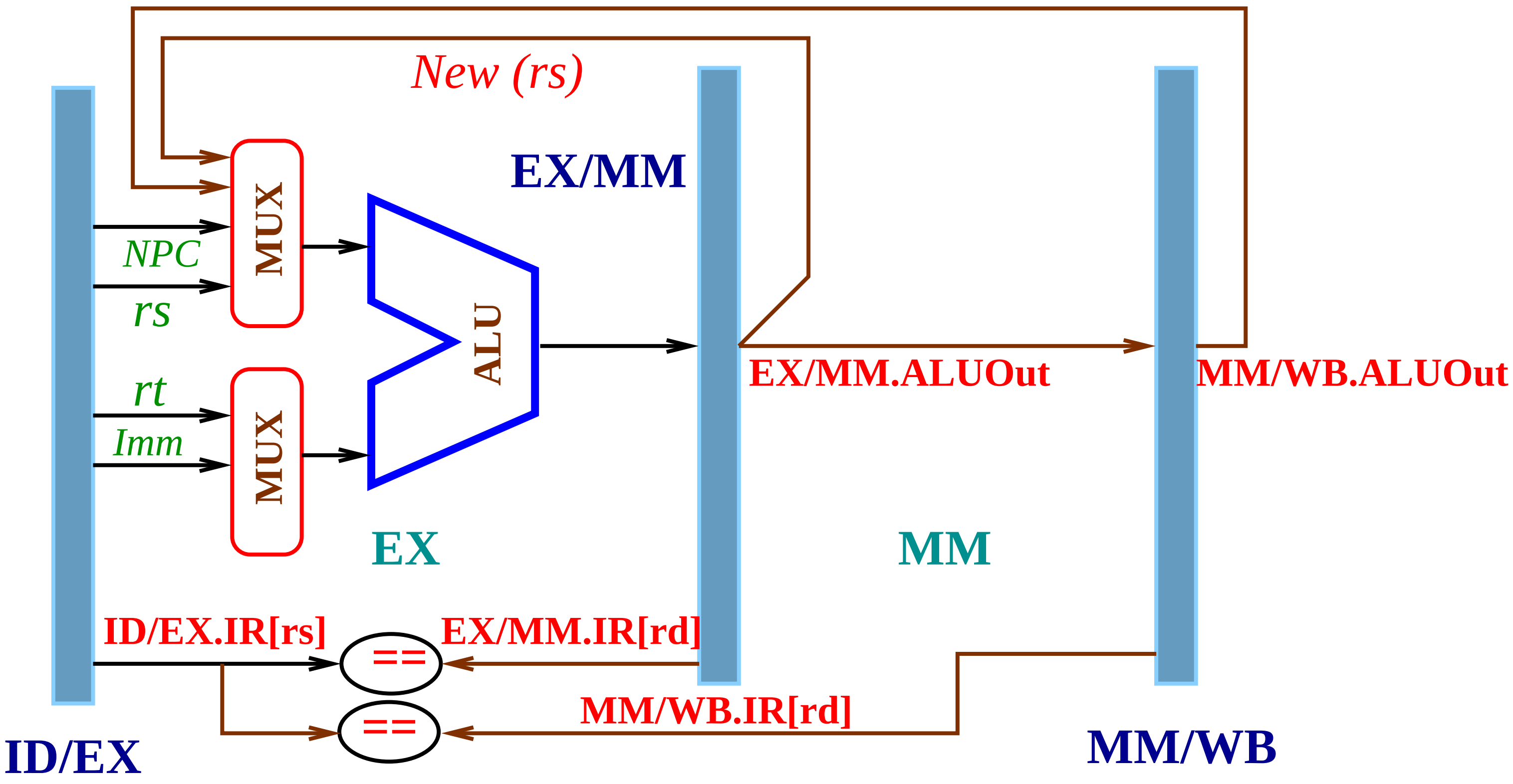

Figure 17: **Forwarding in MIPS Data Path**

A Problem

- Consider the following code:
  DADD R3, R2, R1
  DSUB R3, R3, R5
  DADDI R4, R3, Data.
- The instruction DSUB should use the content of R3 produced by DADD. The forwarding should be from EX/MM.ALUOut.
- The instruction DADDI should use the content of R3 produced by DSUB. The forwarding should be from EX/MM.ALUOut (and not from MM/WB.ALUOut).

Forwarding from MM/WB

- **Source**: LD R3 (rt), off(R2) (rs)
- **Destination**: DADDI R4 (rt), R3 (rs), imm: R4 $\leftarrow$ R3 + imm.
- **Sequence**:
  LD R3, off(R2)
  DADD R5, R2, R1
  DAAI R4, R3, imm.
- **Compare**: MM/WB.IR[rt] == ID/EX.IR[rs].
- The MM/WB.LDM is to be forwarded to the rs-MUX of the ALU.

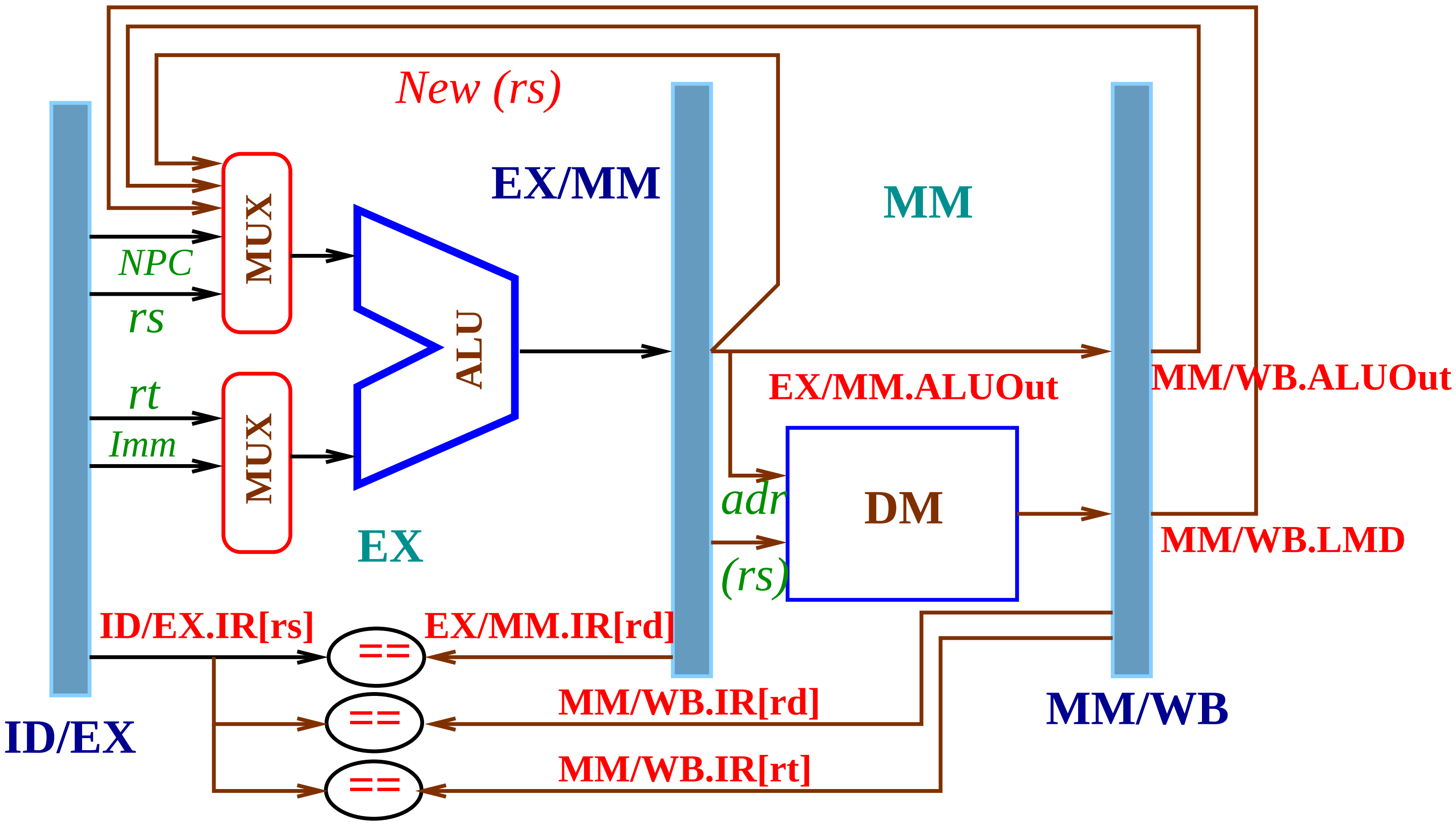

Figure 18: **Forwarding in MIPS Data Path**

Branch in the Pipeline

- **Branch instructions like BEQ and BNE tests equality of two registers: BEQ R1 (rs), R2 (rt), off.**
- **We consider simpler branches like BEQZ and BNEZ: BEQZ R1 (rs), off.**
- **It is possible to complete the decision of branch taken or not at the end of ID cycle by shifting the Zero (0) test from the EX stage.**

Branch in the Pipeline

- The branch address is also computed in the ID stage. But the ALU is active in the EX stage and cannot be shared. So a differrnt adder is introduced.

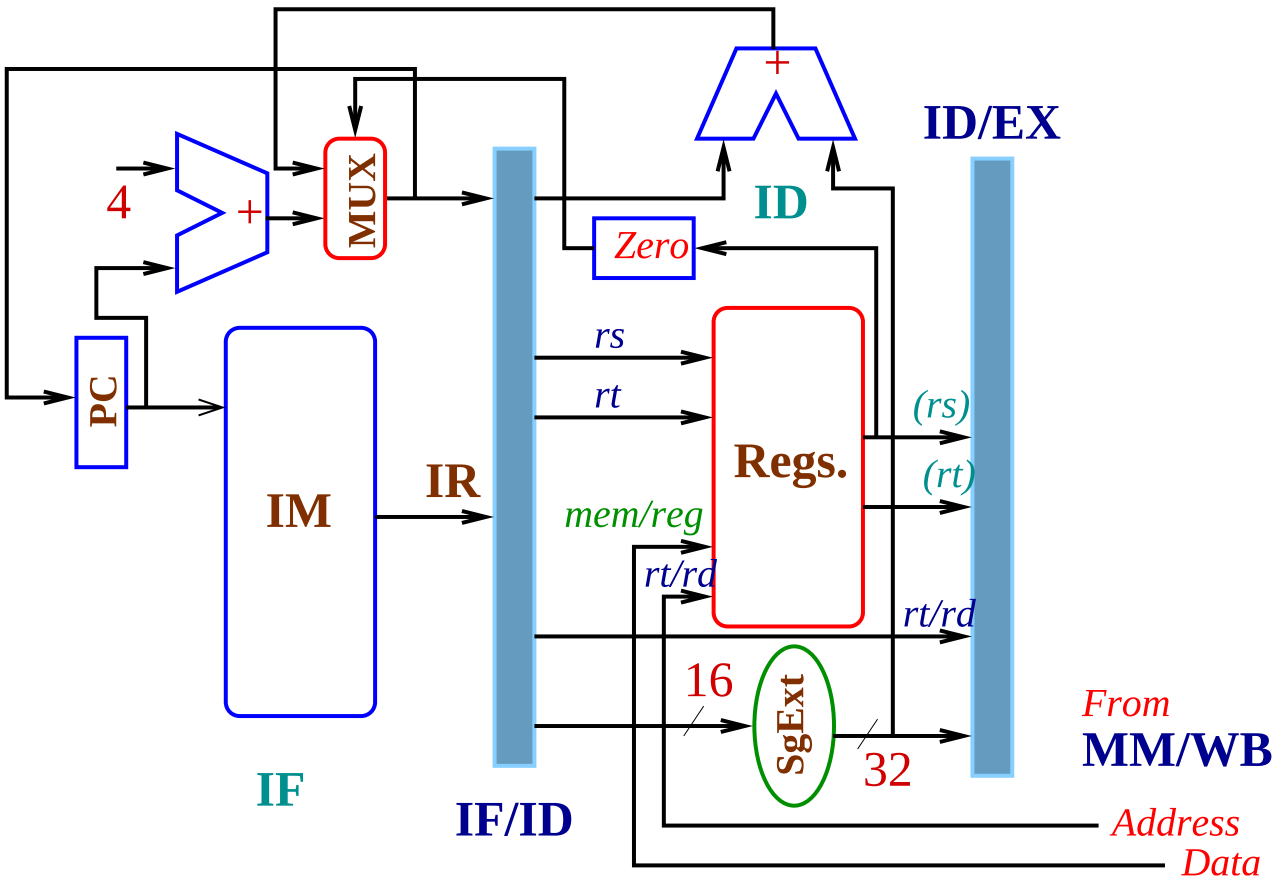

Figure 19: **Modified MIPS Data Path**

With the Modified Data Path

- **There is one (1) clock-cycle delay for a branch taken.**

An Example

Address	Instruction
2000	BEQZ R2, 96
2004	I1
2008	I2
2012	I3
2016	I4
...	...
2100	J1
2104	J2

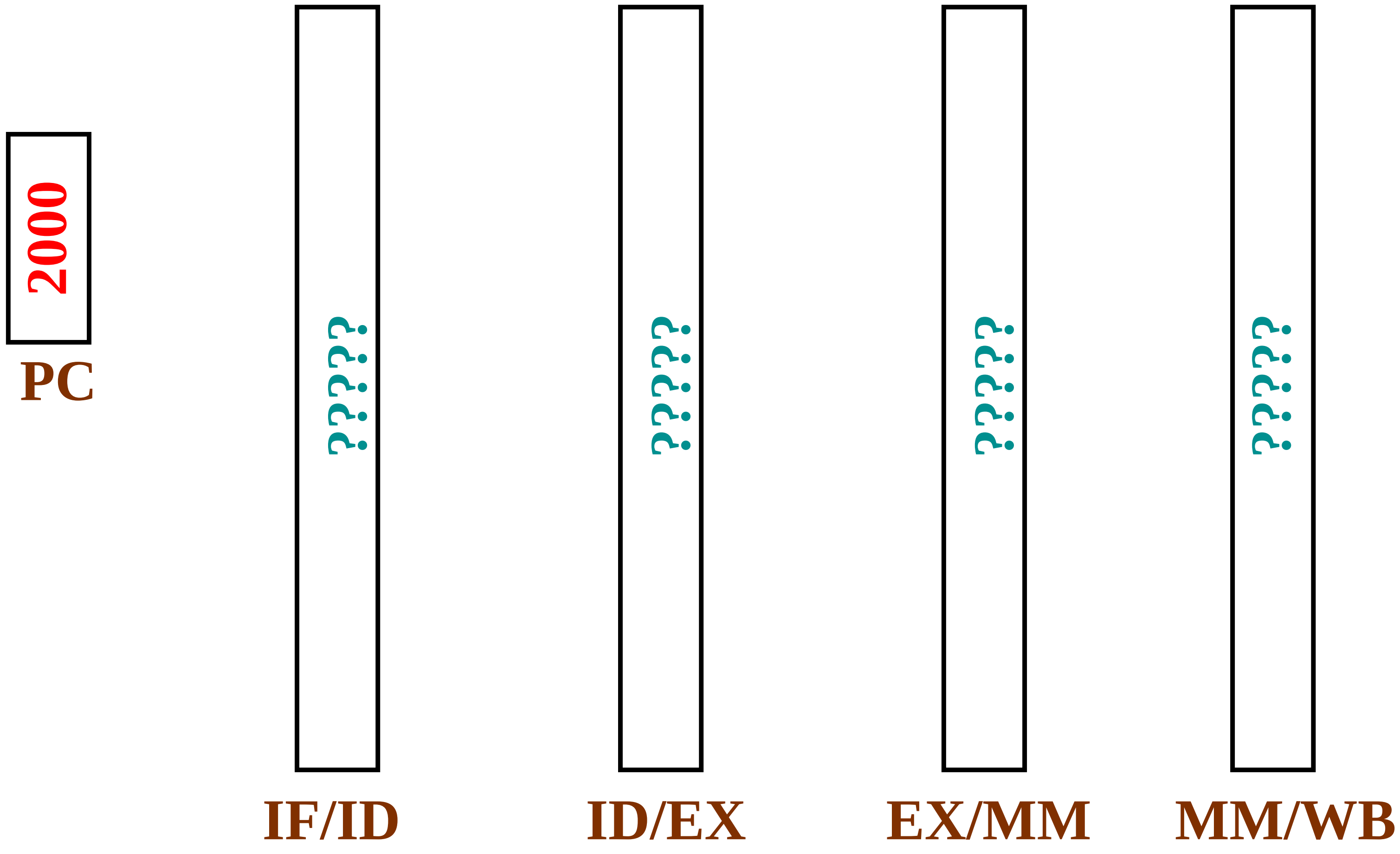

Figure 20: **At the End of t_0**

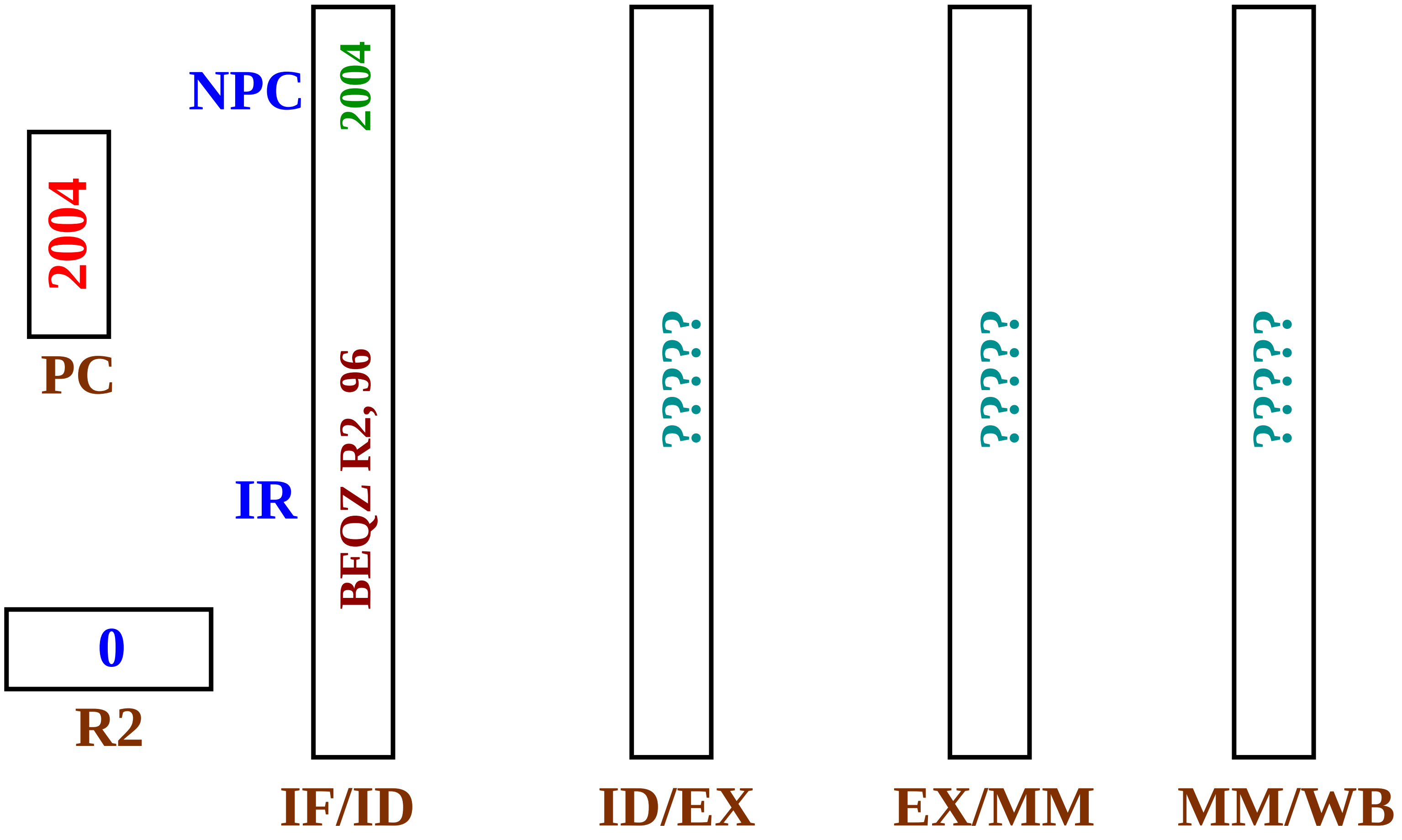

Figure 21: **At the End of $t_0 + 1$**

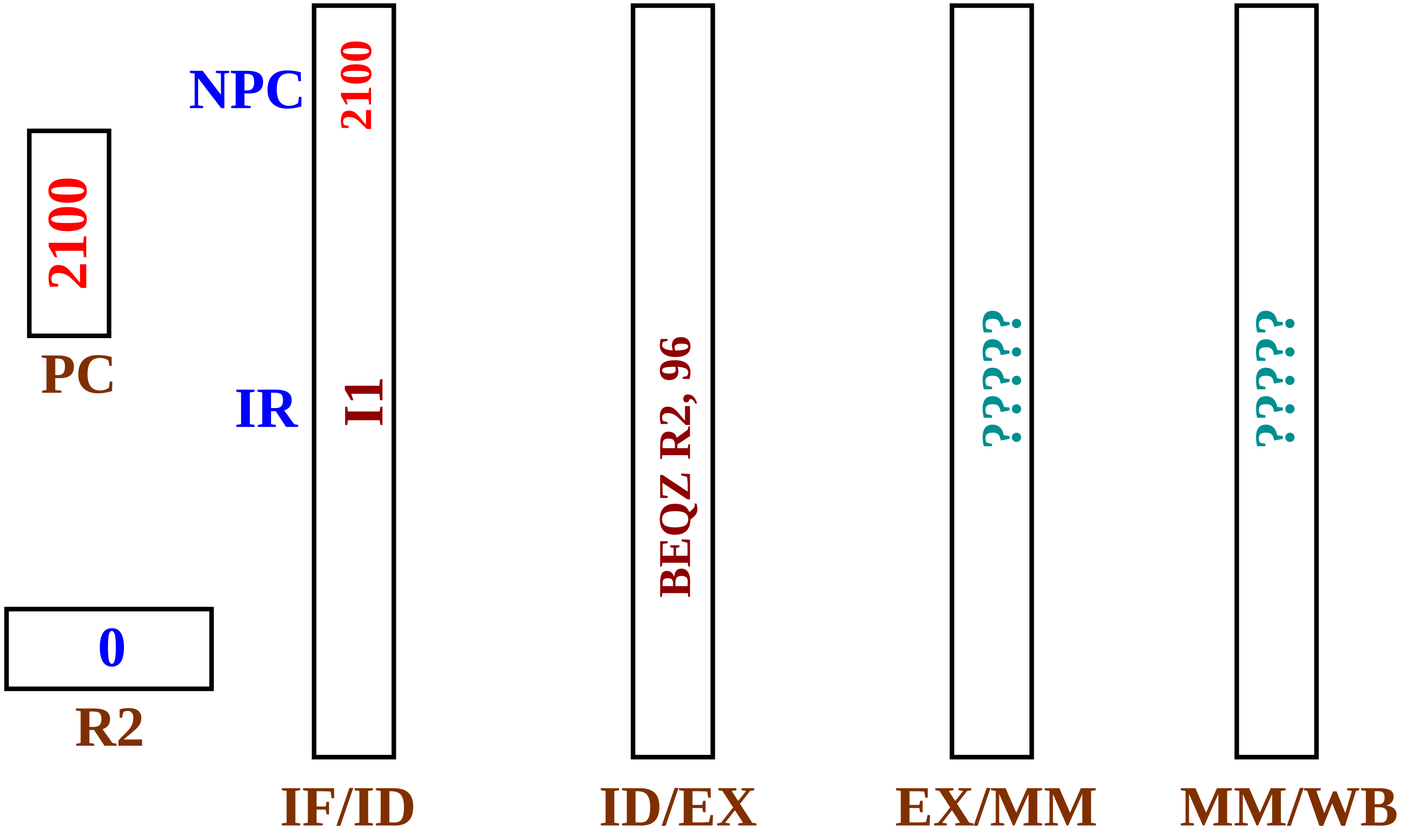

Figure 22: **At the End of $t_0 + 2$**

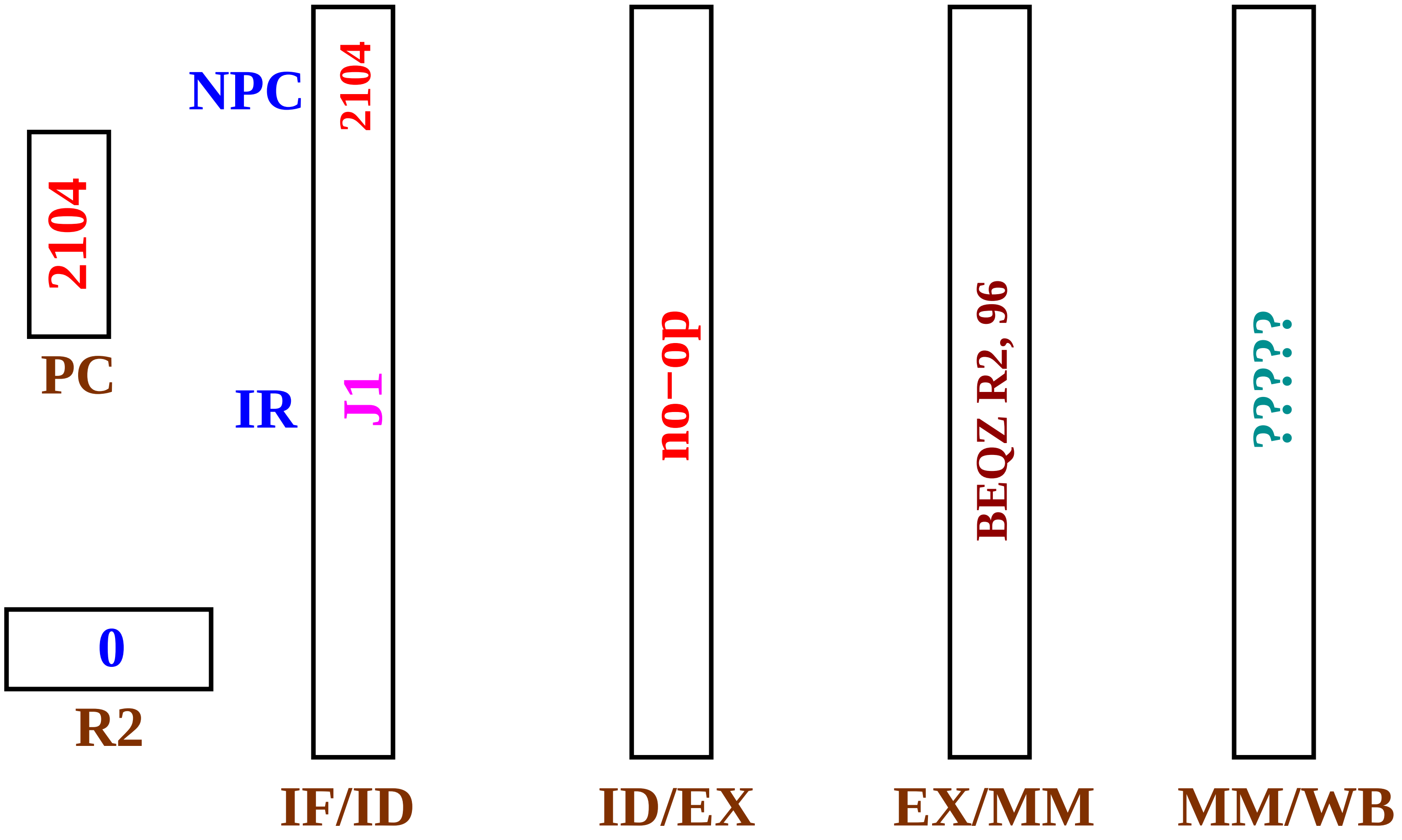

Figure 23: **At the End of $t_0 + 3$**

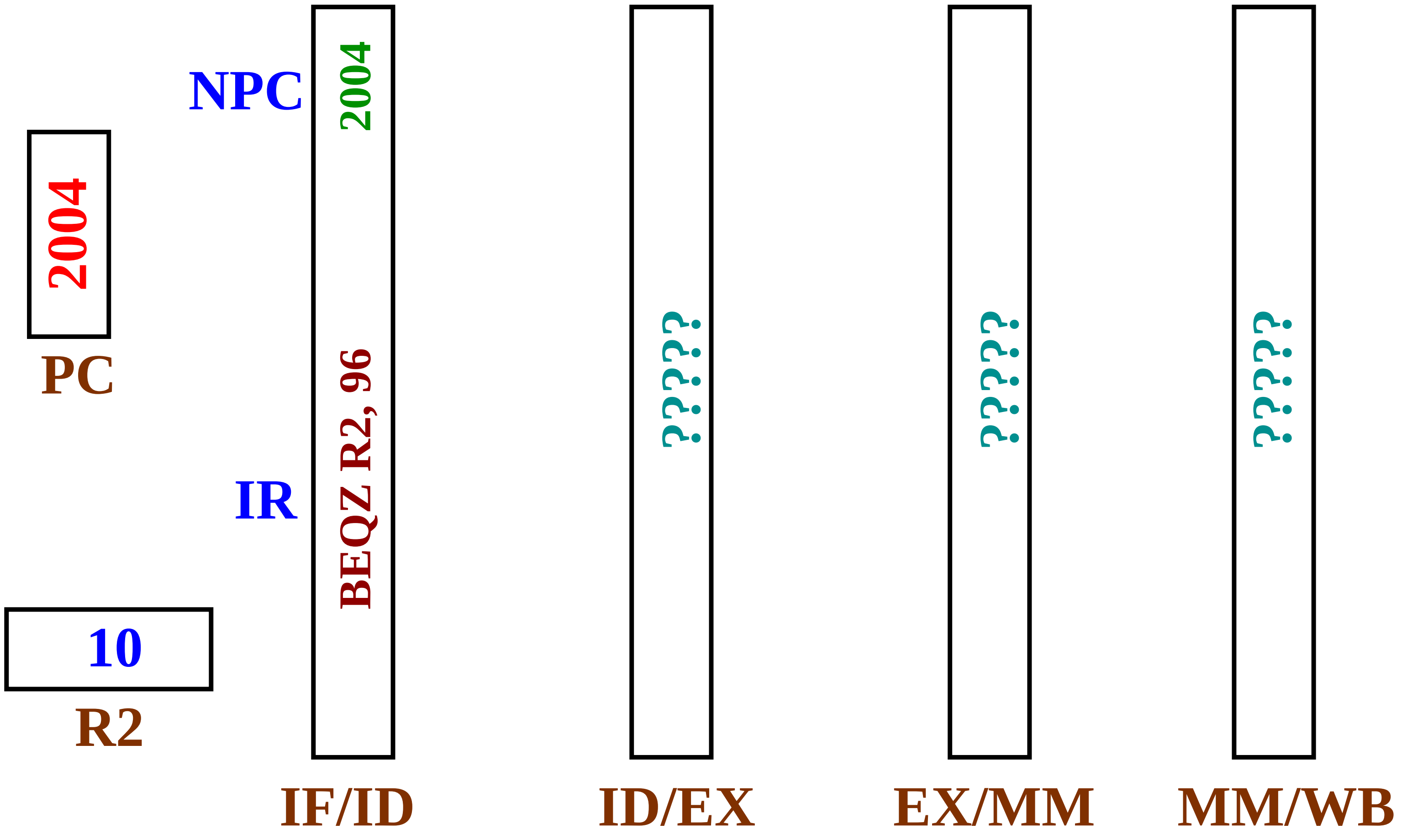

Figure 24: **Not Taken: End of** $t_0 + 1$

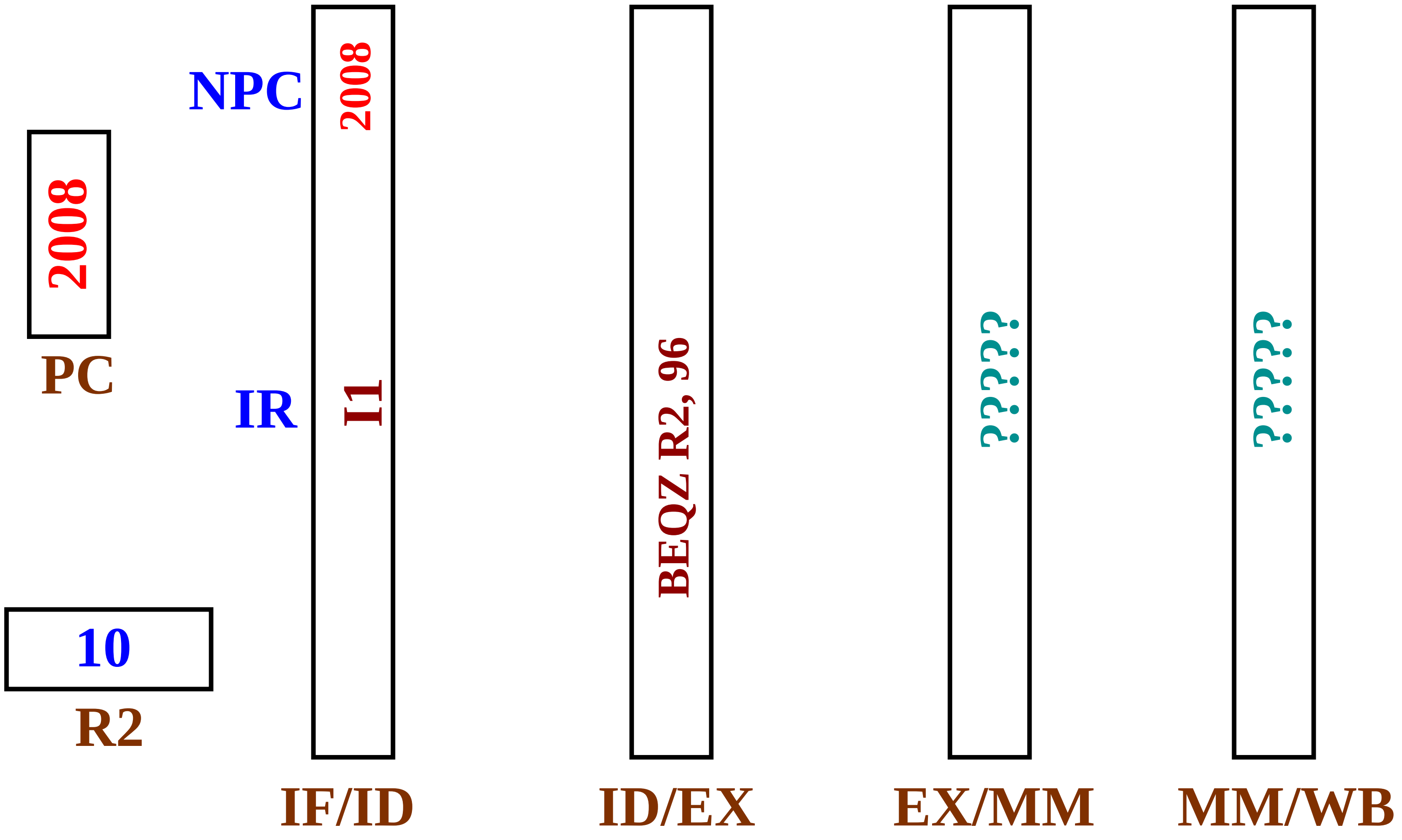

Figure 25: **Not Taken: End of $t_0 + 2$**

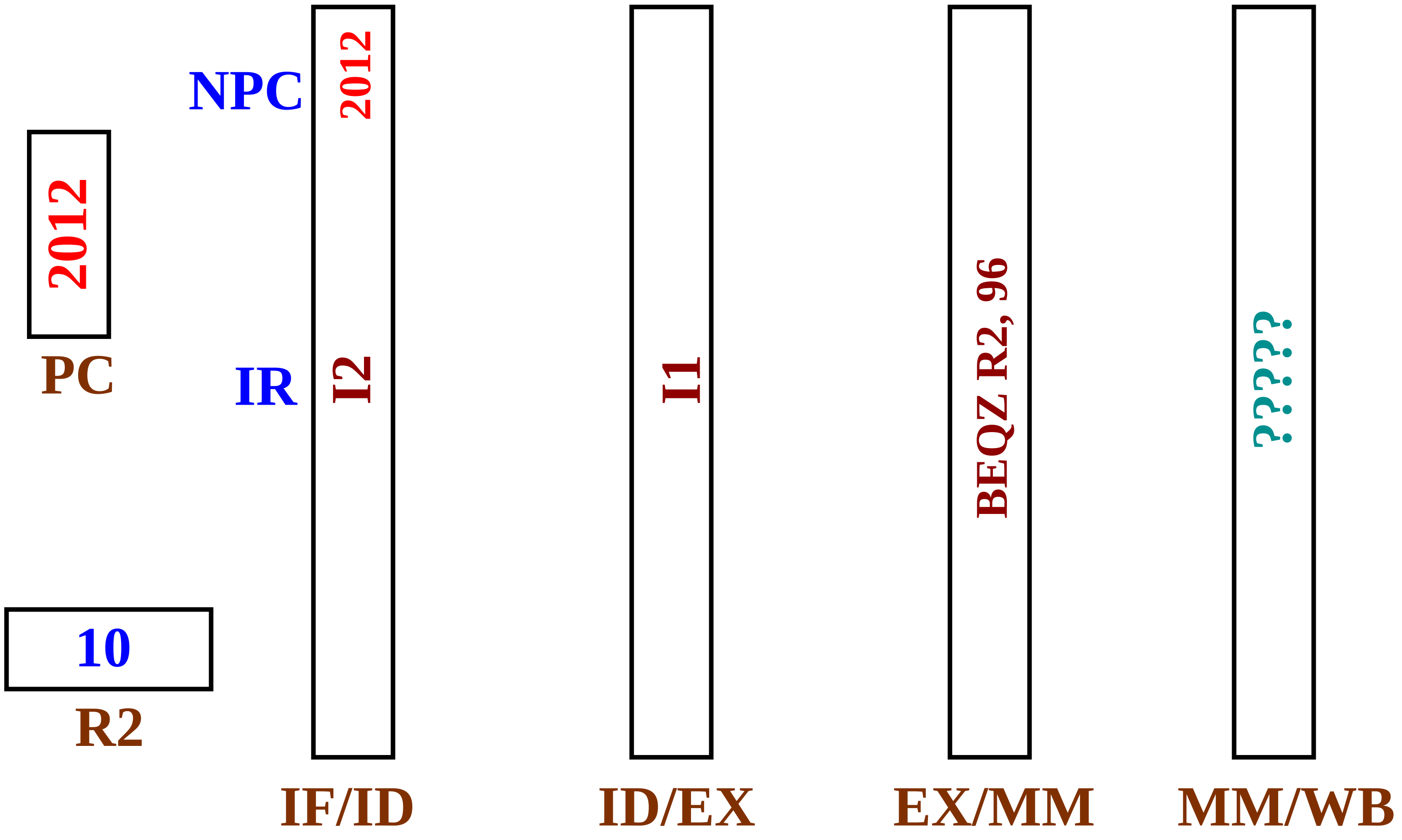

Figure 26: **Not Taken: End of $t_0 + 3$**

Modified Data Path: Branch Taken

	Clock							
Inst.	1	2	3	4	5	6	7	8
Branch	IF	ID	EX	MM	WB			
Br. Tgt.		IF	IF	ID	EX	MM	WB	
Br. Tgt.+1				IF	ID	EX	MM	WB
Br. Tgt.+2					IF	ID	EX	MM
Br. Tgt.+3						IF	ID	EX

Modified Instruction Fetch: IF

```
IF/ID.IR <-- Mem[PC]

{IF/ID.NPC, PC} <-- if((EX/ID.IR[opcode] == branch) &
                        (Reg[IF/ID.IR[rs]] == 0))
                        IF/ID.NPC + SgExt(IF/ID.Imm << 2)
                     else PC + 4
```

Modified Instruction Decode: ID

```
ID/EX.A <-- Reg[IF/ID.IR[rs]]
ID/EX.B <-- Reg[IF/ID.IR[rt]]
```

No need to carry NPC.

```
ID/EX.IR <-- IF/ID.IR
ID/EX.Imm <-- SgExt(IF/ID.IR[imm]])
```

Hazard Due to Modification

- **Consider the following sequence:**
  DSUB R2, R1, R3
  BEQZ R2, offset
- **The value of R2 is not available before the EX cycle and there will be a stall.**

Stall: ALU Inst. before Branch

	Clock							
Inst.	1	2	3	4	5	6	7	8
ALU	IF	ID	EX	MM	WB			
Branch		IF	Stall	ID	EX	MM	WB	

Note · · ·

- **If the computation time for branch condition and the target address is more, the branch delay may be higher.**
- **Deep pipeline or complex instruction set pipeline may have 4-6 clock cycles of delay.**