MIPS Pipelined Implementation

Imp-1

Outline: (In this set.)

Unpipelined Implementation. (Diagram only.)

Pipelined MIPS Implementations: Hardware, notation, hazards.

Dependency Definitions.

Data Hazards: Definitions, stalling, bypassing.

Control Hazards: Squashing, one-cycle implementation.

Outline: (Covered in class but not yet in set.)

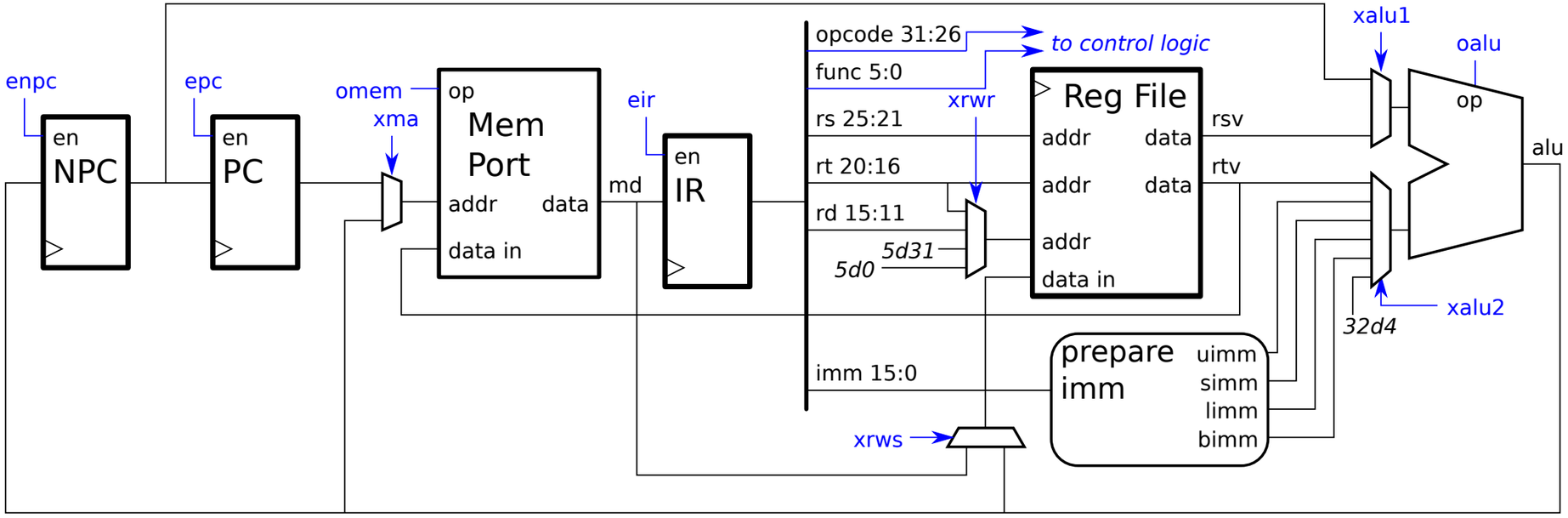
Operation of nonpipelined implementation, elegance and power of pipelined implementation. (See text.)

Computation of CPI for program executing a loop.

Imp-2

Very Simple MIPS Implementation

From EE 3755.



Features

Avoid duplication of hardware: One Memory Port, One Adder (ALU).

Relatively complex control logic needed to re-use ALU, etc.

Imp-2

Unpipelined Implementation

Imp-3

In this implementation hardware *is* duplicated.

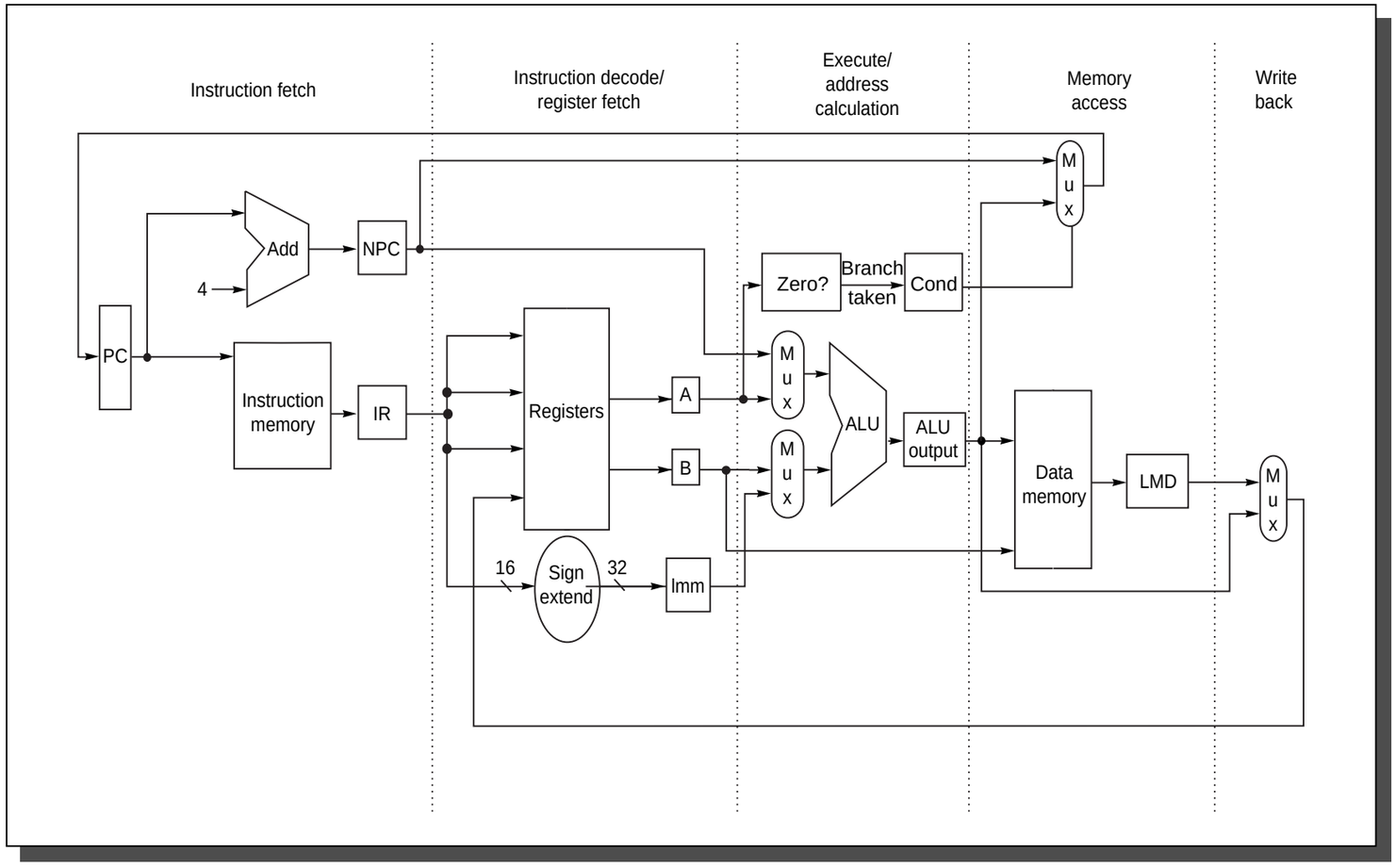
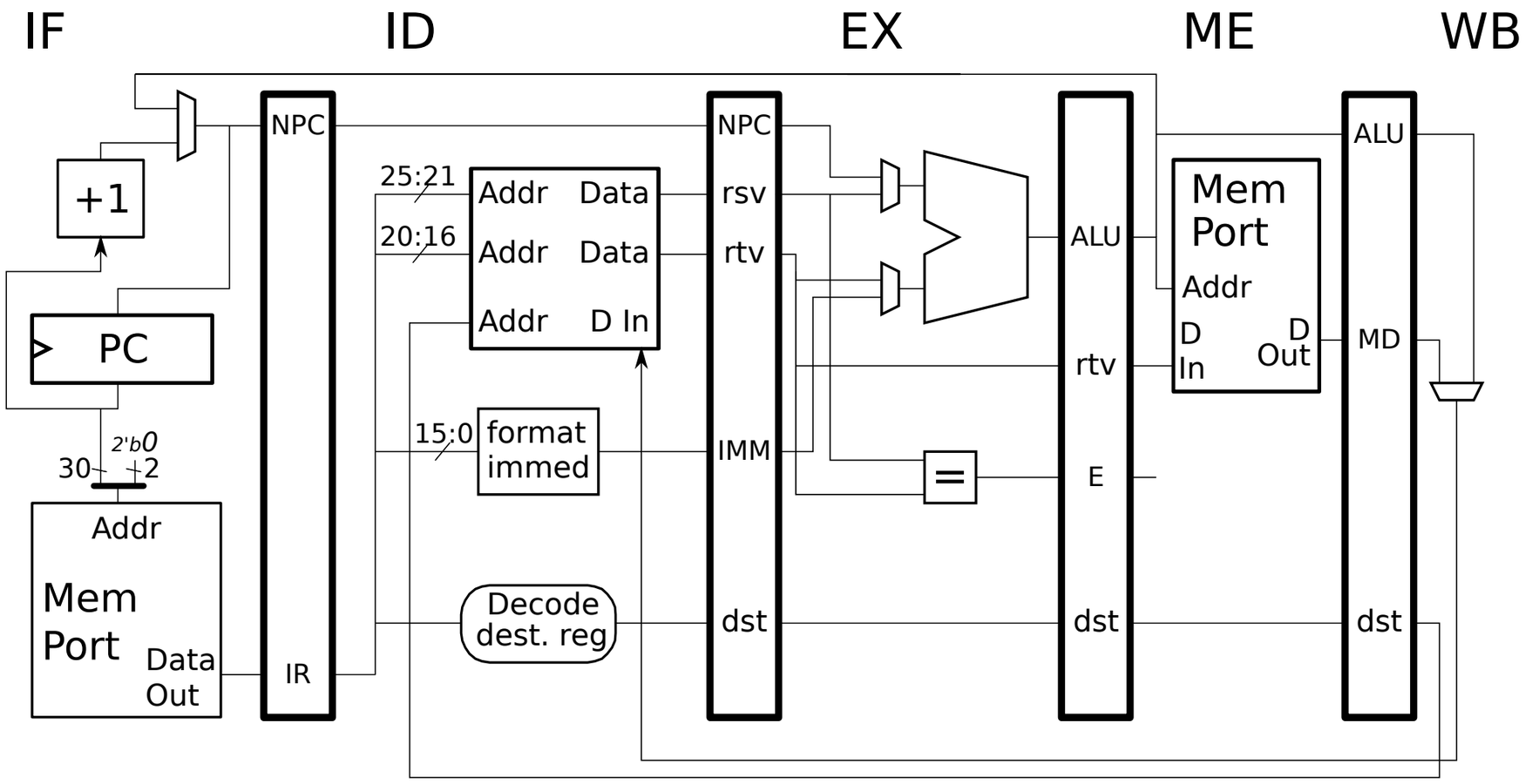


FIGURE 3.1 The implementation of the DLX datapath allows every instruction to be executed in four or five clock cycles.

Imp-3

Pipelined MIPS Implementation

Imp-4



Note: diagram omits connections for some instructions.

LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-4

Pipeline Details

Imp-5

Pipelining Idea

Split hardware into n equally sized (in time) *stages* ...
... separate the stages using special registers called *pipeline latches* ...
... increase the clock frequency by $\lessapprox n\times$ ...
... avoid problems due to overlapping of execution.

Pipeline Stages

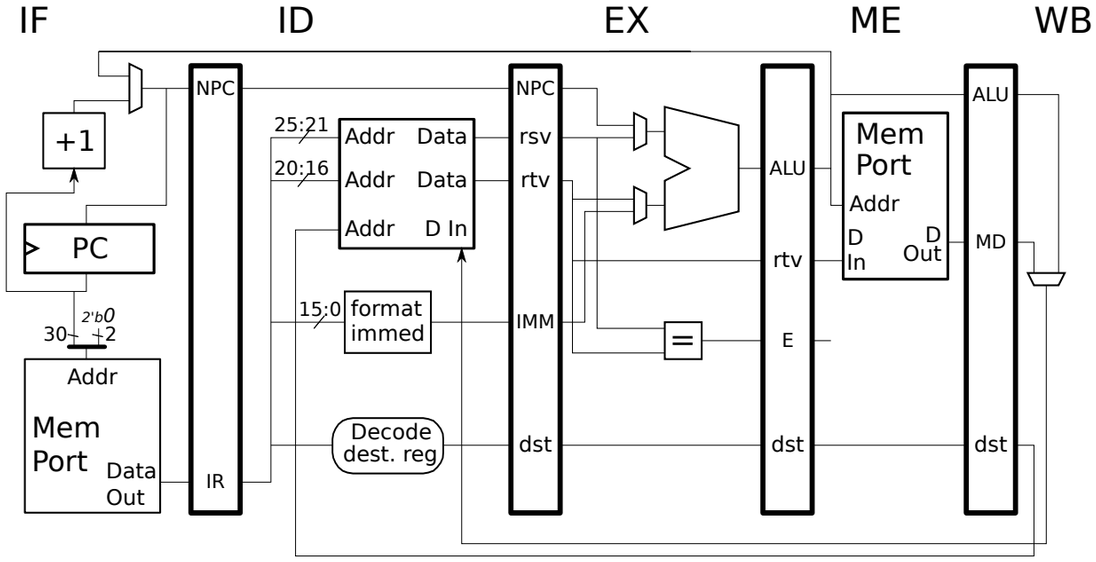
Pipeline divided into *stages*.

Each stage occupied by at most one instruction.

At any time, each stage can be occupied by its own instruction.

Stages given names: `IF, ID, EX, ME, WB`

Sometimes `ME` written as `MEM`.



Imp-5

Imp-6

Pipeline Latches:

Registers separating pipeline stages.

Written at end of each cycle.

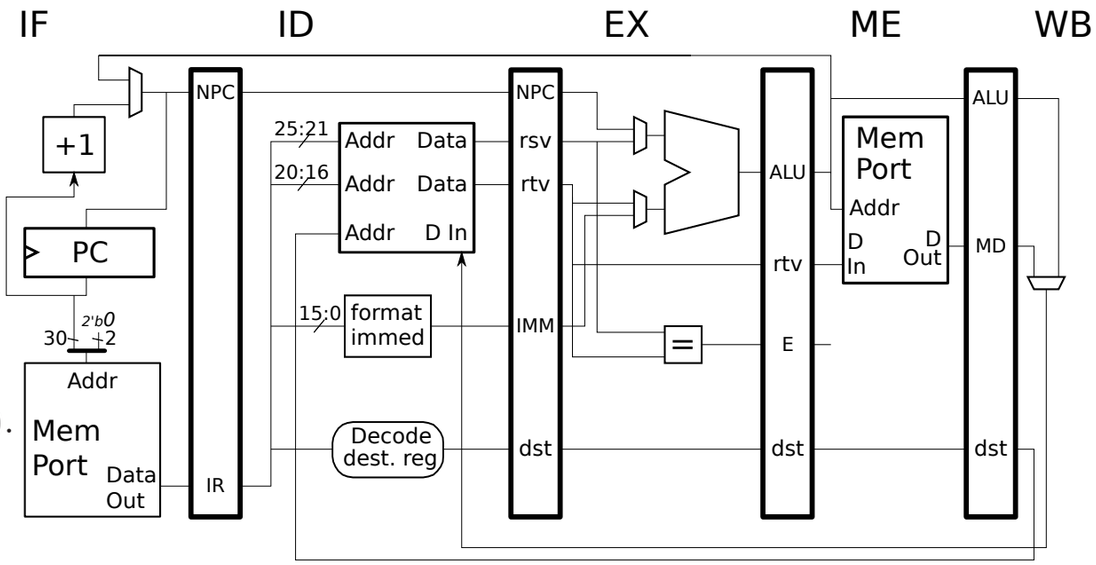
To emphasize role shown in diagram as bar separating stages.

Registers named using pair of stage names and register name.

For example, `IF/ID.IR`, `ID/EX.dst`, `ID/EX.rsv` (used in text, notes).

For brevity first stage name dropped: `ID.IR`, `EX.dst`, `EX.rsv`.

`if_id_ir`, `id_ex_ir`, `id_ex_rs_val` (used in Verilog code).



LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.
Imp-6

Pipeline Execution Diagram

Imp-7

Pipeline Execution Diagram:

Diagram showing the pipeline stages that instructions occupy as they execute.

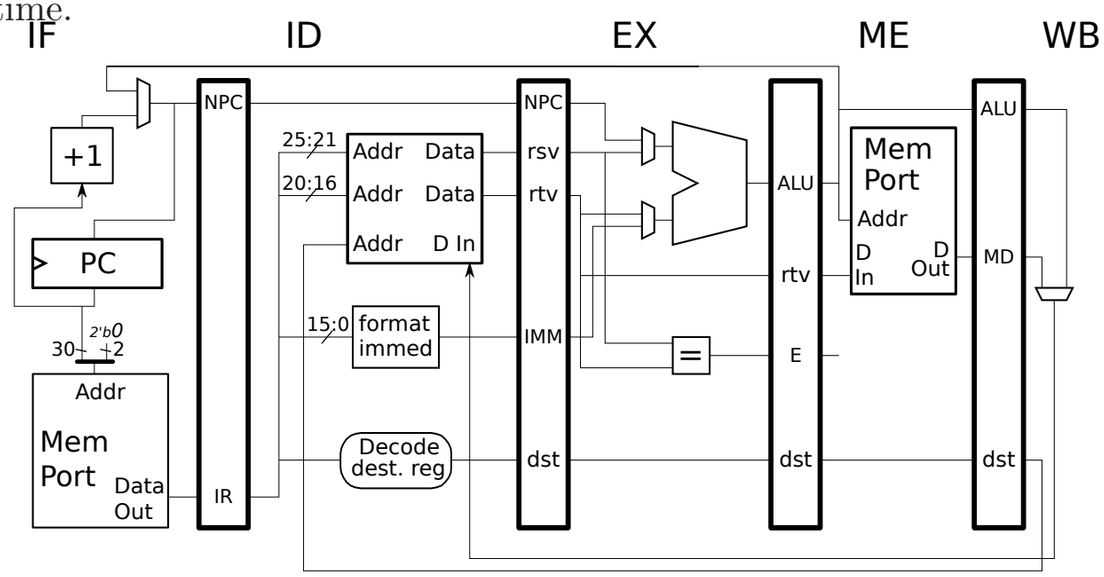
Time on horizontal axis, instructions on vertical axis.

Diagram shows where instruction is at a particular time.

```
# Cycle          0   1   2   3   4   5   6
add r1, r2, r3   IF  ID  EX  ME  WB
and r4, r5, r6       IF  ID  EX  ME  WB
lw  r7, 8(r9)            IF  ID  EX  ME  WB
```

A vertical slice (*e.g.* /, at cycle 3) shows processor activity at that time.

In such a slice **a stage should appear at most once** . . .
. . . if it appears more than once execution not correct . . .
. . . since a stage can only execute one instruction at a time.



LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-7

Instruction Decoding and Pipeline Control

Imp-8

Pipeline Control

Setting control inputs to devices including . . .

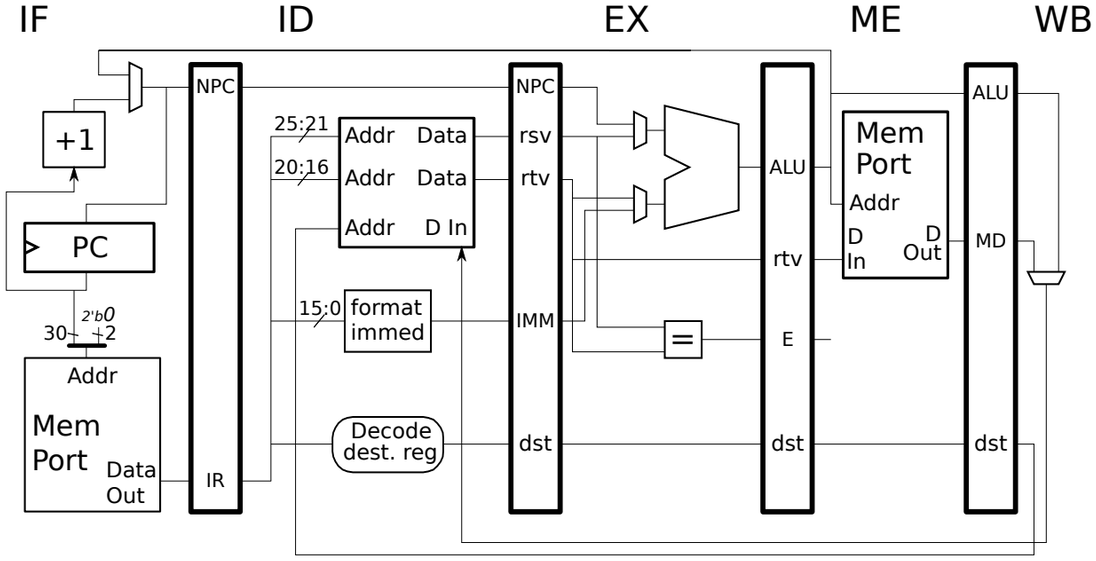
. . . multiplexor inputs . . .

. . . function for ALU . . .

. . . operation for memory . . .

. . . whether to clock each register . . .

. . . *et cetera.*



LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-8

Imp-9

Options for controlling pipeline:

- Decode in ID

  Determine settings in ID, pass settings along in pipeline latches.

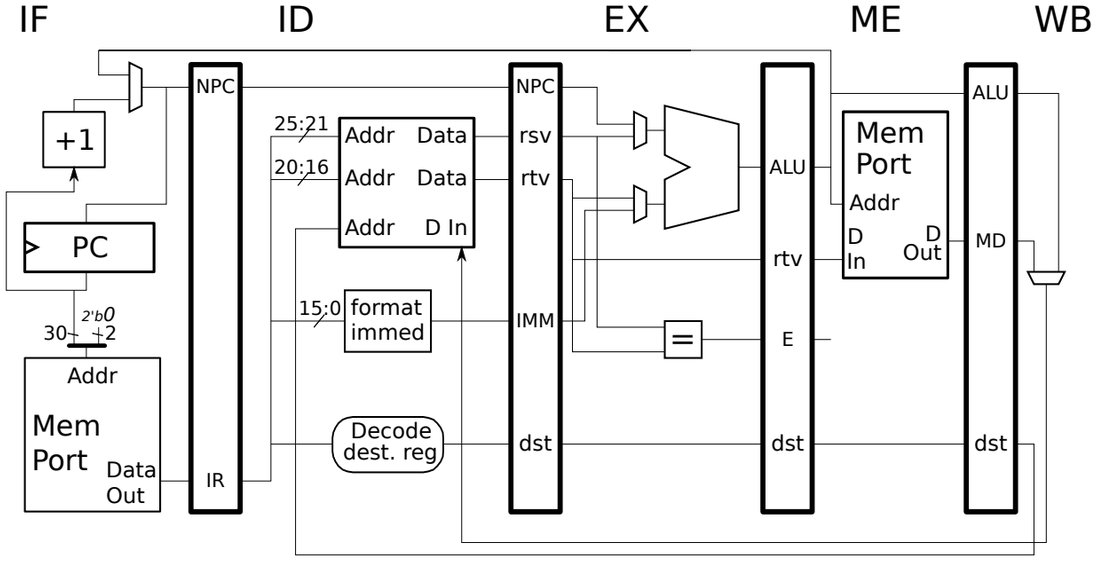
- Decode in Each Stage

  Pass opcode portions of instruction along.

  Decoding performed as needed.

Real systems decode in ID.

Example given later in this set.



Imp-9

Dependencies and Hazards

Imp-10

Remember

Operands **read from** registers in ID. . .
. . . and results **written to** registers in WB.

Consider the following **incorrect execution**:

```
# Cycle          0   1   2   3   4   5   6   7
add  r1, r2, r3  IF  ID  EX  ME  WB
sub  r4, r1, r5      IF  ID  EX  ME  WB
and  r6, r1, r8          IF  ID  EX  ME  WB
xor  r9, r4, r11             IF  ID  EX  ME  WB
```

Execution incorrect because . . .
. . . `sub` reads `r1` before `add` writes (or even finishes computing) r1, . . .
. . . `and` reads `r1` before `add` writes r1, and . . .
. . . `xor` reads `r4` before `sub` writes r4.

LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-10

Dependencies and Hazards

Imp-11

Incorrect execution due to...
... *dependencies* in program...
... **and** *hazards* in hardware (pipeline).

Incorrect execution above is the "fault" of the hardware...
... because the ISA does not forbid dependencies.

Dependency:
A relationship between two instructions ...
... indicating that their execution should be (or appear to be) in program order.

Hazard:
A potential execution problem in an implementation due to overlapping instruction execution.

There are several kinds of dependencies and hazards.

For each kind of dependence there is a corresponding kind of hazard.

Imp-11

Dependencies

Imp-12

Dependency:
A relationship between two instructions . . .
. . . indicating that their execution should be, or appear to be, in program order.

If B is dependent on A then B should appear to execute after A.

Dependency Types:

- *True, Data, or Flow Dependence* (Three different terms used for the same concept.)
- *Name Dependence*
- *Control Dependence*

Imp-12

Data Dependence

Imp-13

Data Dependence: (a.k.a., *True* and *Flow* Dependence)
A dependence between two instructions . . .
. . . indicating data needed by the second is produced by the first.

Example:

```
add  r1, r2, r3
sub  r4, r1, r5
and  r6, r4, r7
```

The `sub` is dependent on `add` (via `r1`).

The `and` is dependent on `sub` (via `r4`).

The `and` is dependent `add` (via `sub`).

Execution may be incorrect if . . .
. . . a program having a data dependence . . .
. . . is run on a processor having an uncorrected RAW hazard.

Imp-13

Name Dependencies

Imp-14

There are two kinds: *antidependence* and *output dependence*.

Antidependence:
A dependence between two instructions ...
... indicating a value written by the second ...
... that the first instruction reads.

Antidependence Example

```
add  r1, r2, r3
sub  r2, r4, r5
```

`sub` is antidependent on the `add`.

Execution may be incorrect if ...
... a program having an antidependence ...
... is run on a processor having an uncorrected WAR hazard.

Imp-14

Imp-15

Output Dependence:

A dependence between two instructions ...
... indicating that both instructions write the same location ...
... (register or memory address).

Output Dependence Example

```
add  r1, r2, r3
sub  r1, r4, r5
```

The `sub` is output dependent on `add`.

Execution may be incorrect if ...
... a program having an output dependence ...
... is run on a processor having an uncorrected WAW hazard.

Imp-15

Imp-16

Control Dependence:

A dependence between a branch instruction and a second instruction ...

... indicating that whether the second instruction executes ...

... depends on the outcome of the branch.

```
 beq  $1, $0 SKIP   # Delayed branch
 nop
 add   $2, $3, $4
SKIP:
 sub   $5, $6, $7
```

The `add` is control dependent on the `beq`.

The `sub` is not control dependent on the `beq`.

Imp-16

Pipeline Hazards

Imp-17

Hazard:
A potential execution problem in an implementation due to overlapping instruction execution.

Interlock:
Hardware that avoids hazards by stalling certain instructions when necessary.

Hazard Types:

Structural Hazard:
Needed resource currently busy.

Data Hazard:
Needed value not yet available or overwritten.

Control Hazard:
Needed instruction not yet available or wrong instruction executing.

Imp-17

Data Hazards

Imp-18

Identified by acronym indicating correct operation.

- *RAW:* Read after write, akin to data dependency.
- *WAR:* Write after read, akin to anti dependency.
- *WAW:* Write after write, akin to output dependency.

MIPS implementations above only subject to RAW hazards.

RAR not a hazard since read order irrelevant (without an intervening write).

Interlocks

Imp-19

When threatened by a hazard:

- *Stall* (Pause a part of the pipeline.)
  Stalling avoids overlap that would cause error.

  This does slow things down.

- Add hardware to avoid the hazards.
  Details of hardware depend on hazard and pipeline.

  Several will be covered.

Imp-19

Structural Hazards

Imp-20

Cause: two instructions simultaneously need one resource.

Solutions:

Stall.

Duplicate resource.

Pipelines in this section **do not** have structural hazards.

Covered in more detail with floating-point instructions.

Data Hazards

Imp-21

Pipelined MIPS Subject to RAW Hazards.

Consider the following **incorrect execution** of code containing data dependencies.

```
# Cycle          0   1   2   3   4   5   6   7
add  r1, r2, r3  IF  ID  EX  ME  WB
sub  r4, r1, r5      IF  ID  EX  ME  WB
and  r6, r1, r8          IF  ID  EX  ME  WB
xor  r9, r4, r11             IF  ID  EX  ME  WB
```

Execution incorrect because . . .

. . . `sub` reads `r1` before `add` writes (or even finishes computing) r1, . . .

. . . `and` reads `r1` before `add` writes r1, and . . .

. . . `xor` reads `r4` before `sub` writes r4.

Problem fixed by *stalling* the pipeline.

LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-21

Imp-22

Stall:
To pause execution in a pipeline from IF up to a certain stage.

With stalls, code can execute correctly:

For code on previous slide, stall until data in register.

```
# Cycle          0   1   2   3   4   5   6   7   8   9   10
add  r1, r2, r3  IF  ID  EX  ME  WB
sub  r4, r1, r5      IF  ID  ----->  EX  ME  WB
and  r6, r1, r8          IF  ----->  ID  EX  ME  WB
xor  r9, r4, r11                     IF  ID  ->  EX  ME  WB
```

Arrow shows that instructions stalled.

Stall creates a *bubble*, stages without valid instructions, in the pipeline.

With bubbles present, CPI is greater than its ideal value of 1.

LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.
Imp-22

Imp-23

Stall Implementation

Stall implemented by asserting a *hold* signal ...
... which inserts a `nop` (or equivalent) after the stalling instruction ...
... and disables clocking of pipeline latches before the stalling instruction.

```
# Cycle          0   1   2   3   4   5   6   7   8   9   10
add  r1, r2, r3  IF  ID  EX  ME  WB
sub  r4, r1, r5      IF  ID  ----->  EX  ME  WB
and  r6, r1, r8          IF  ----->  ID  EX  ME  WB
xor  r9, r4, r11                     IF  ID  ->  EX  ME  WB
```

During cycle 3, a `nop` is in EX.

During cycle 4, a `nop` is in EX and ME .

The two adjacent `nop`s are called a *bubble* ...
... they move through the pipeline with the other instructions.

A third `nop` is in EX in cycle 7.

LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.
Imp-23

Bypassing

Imp-24

Some stalls are avoidable.

Consider again:

```
# Cycle          0   1   2   3   4   5   6   7   8   9   10
add  r1, r2, r3  IF  ID  EX  ME  WB
sub  r4, r1, r5      IF  ID  EX  ME  WB
and  r6, r1, r8          IF  ID  EX  ME  WB
xor  r9, r4, r11             IF  ID  EX  ME  WB
```

Note that the new value of `r1` needed by `sub` . . .
. . . has been computed at the end of cycle 2 . . .
. . . and isn't really needed until the beginning of the *next* cycle, 3.

Execution was incorrect because the value had to go around the pipeline to ID.

Why not provide a shortcut?

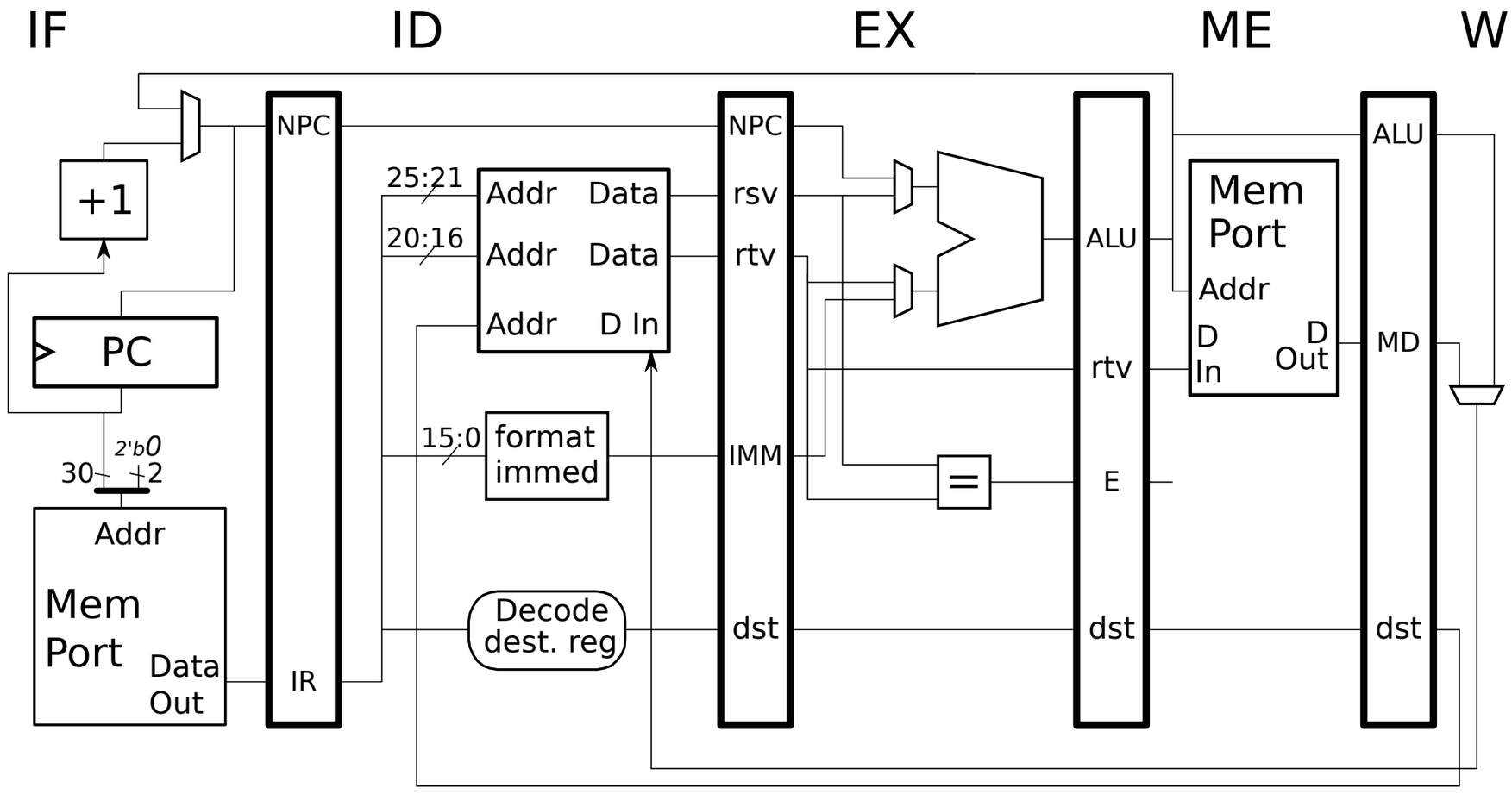
Why not call a shortcut a *bypass* or *forwarding* path?

LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-24

Non-Bypassed MIPS

Imp-25

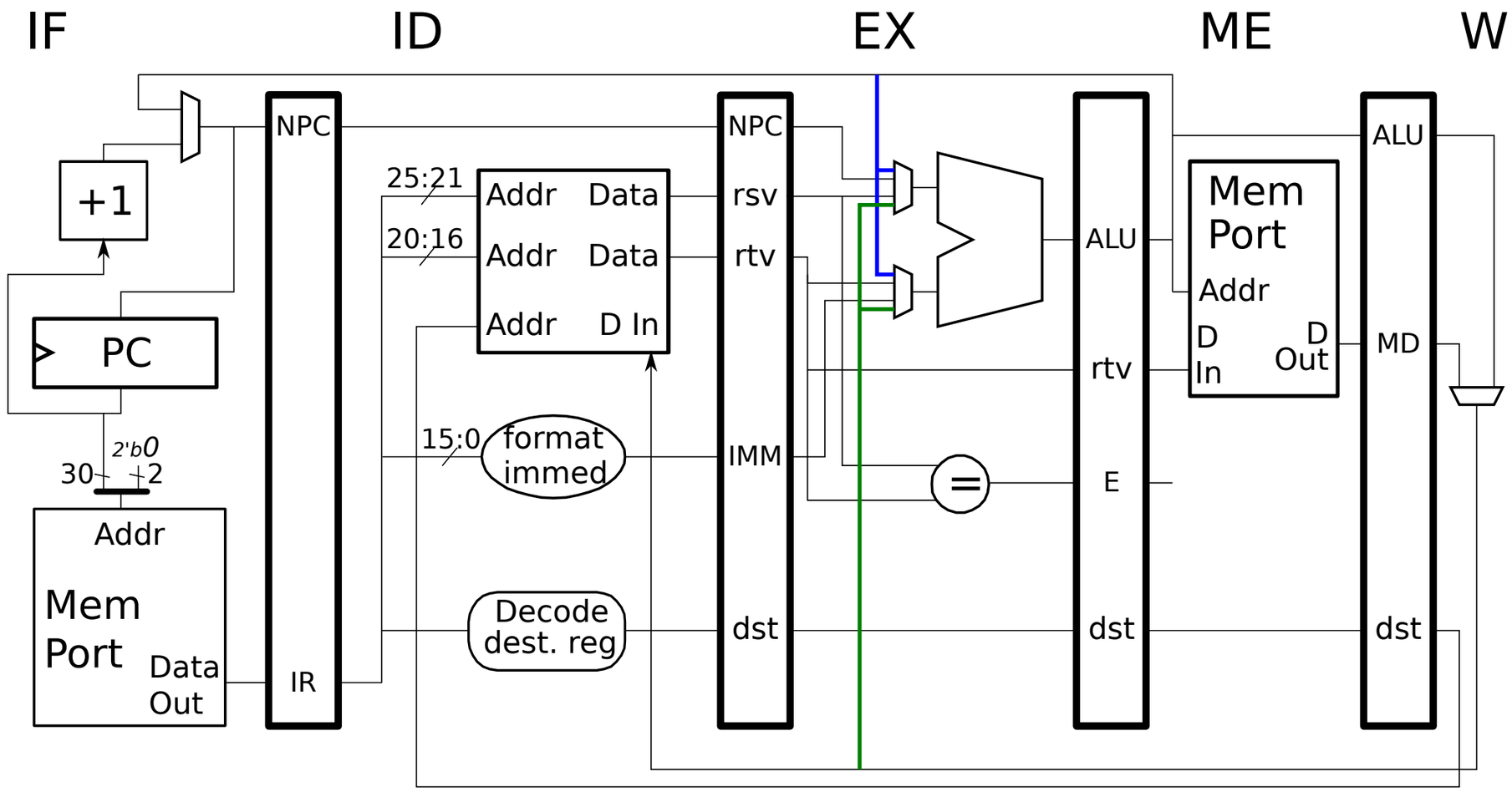


LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-25

Bypassed MIPS

Imp-26

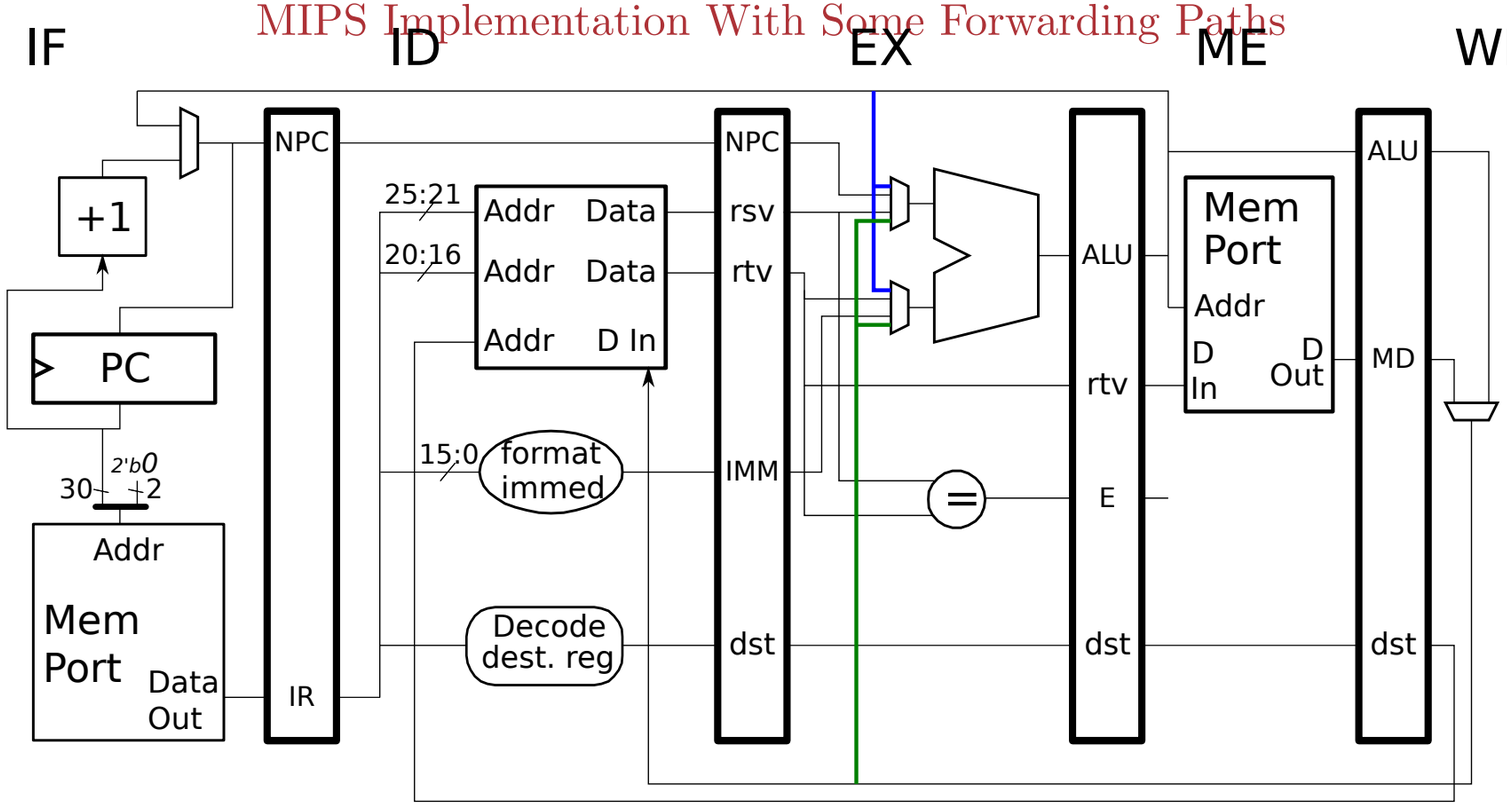


LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-26

MIPS Implementation With Some Forwarding Paths

Imp-27




```
# Cycle          0   1   2   3   4   5   6   7   8   9   10
add   r1, r2, r3 IF  ID  EX  ME  WB
sub   r4, r1, r5     IF  ID  EX  ME  WB
and   r6, r1, r8         IF  ID  EX  ME  WB
xor   r9, r4, r11            IF  ID  EX  ME  WB
```

It works!

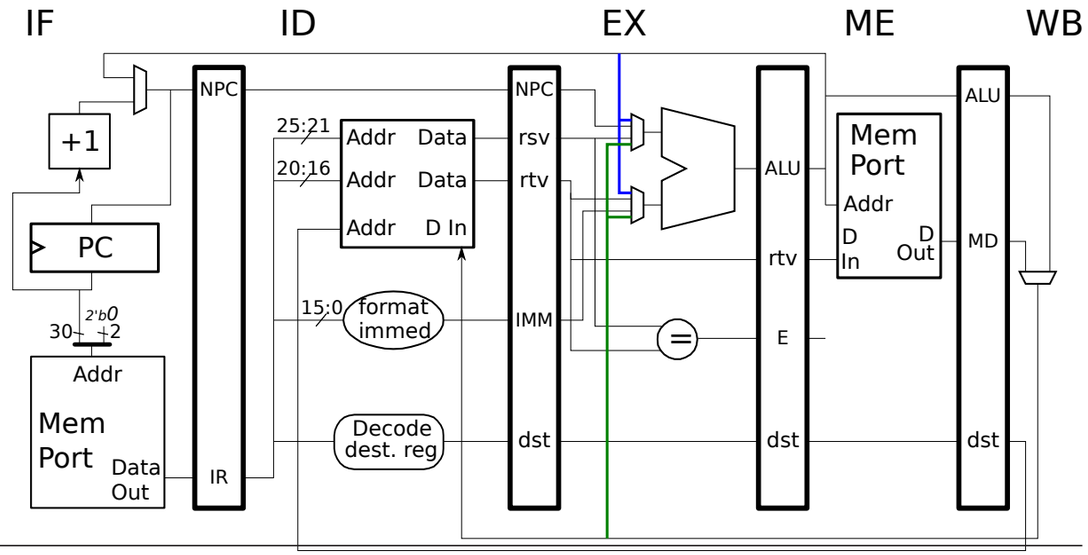
LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-27

MIPS Implementation With Some Forwarding Paths

Imp-28

Some stalls unavoidable.




```
# Cycle          0   1   2   3   4   5   6   7   8   9   10
lw   r1, 0(r2)   IF  ID  EX  ME  WB
add  r1, r1, r4      IF  ID  ->  EX  ME  WB
sw   4(r2), r1           IF  ->  ID  ----->  EX  ME  WB
addi r2, r2, 8               IF  ----->  ID  EX  ME  WB
```

Stall due to `lw` could not be avoided (data not available in cycle 3).

Stall in cycles 5 and 6 could be avoided with a new forwarding path.

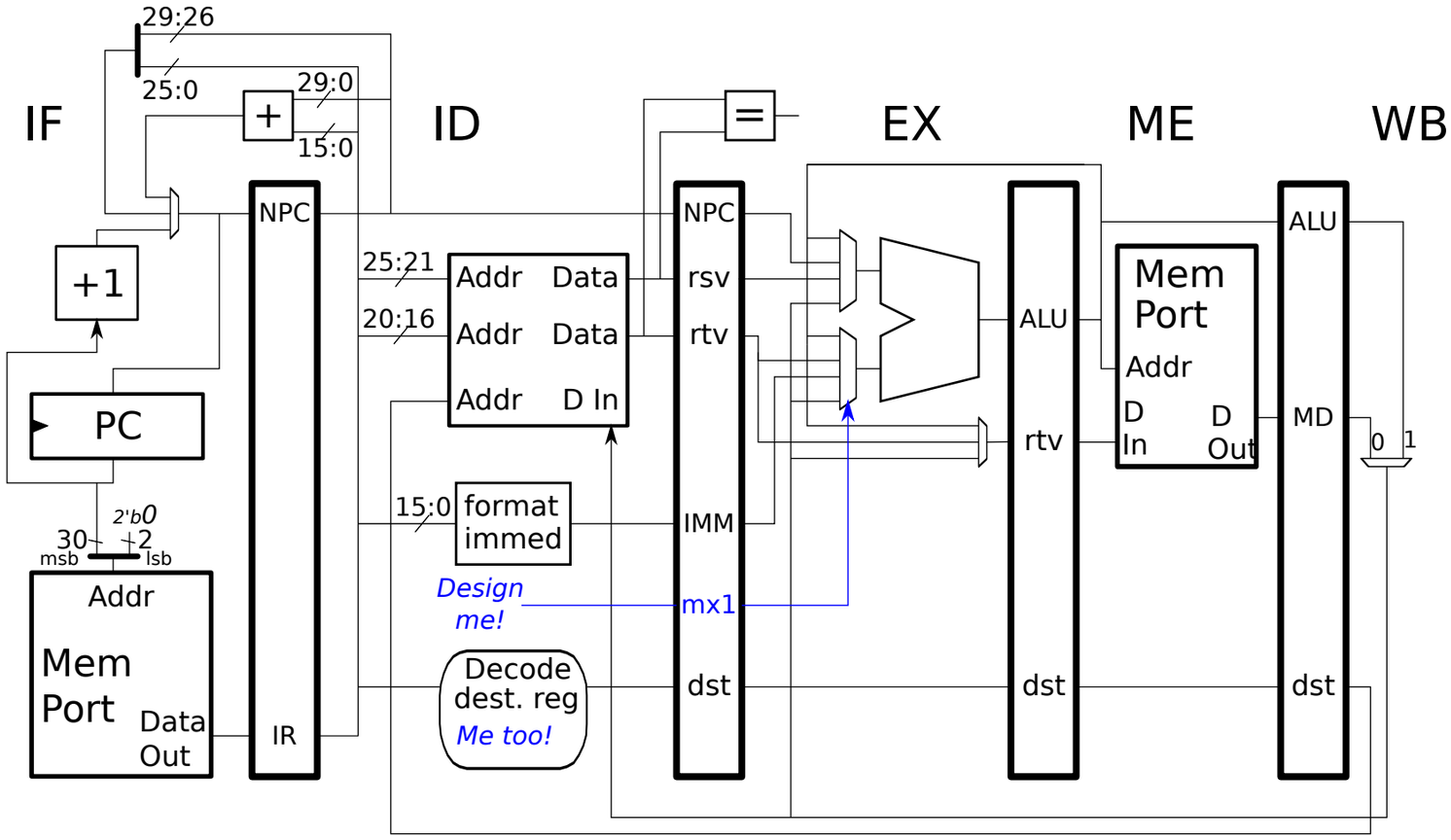
LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-28

Bypass Control Logic for Lower ALU Mux

Imp-29

Start with logic for `dst`, show path of Mux logic.

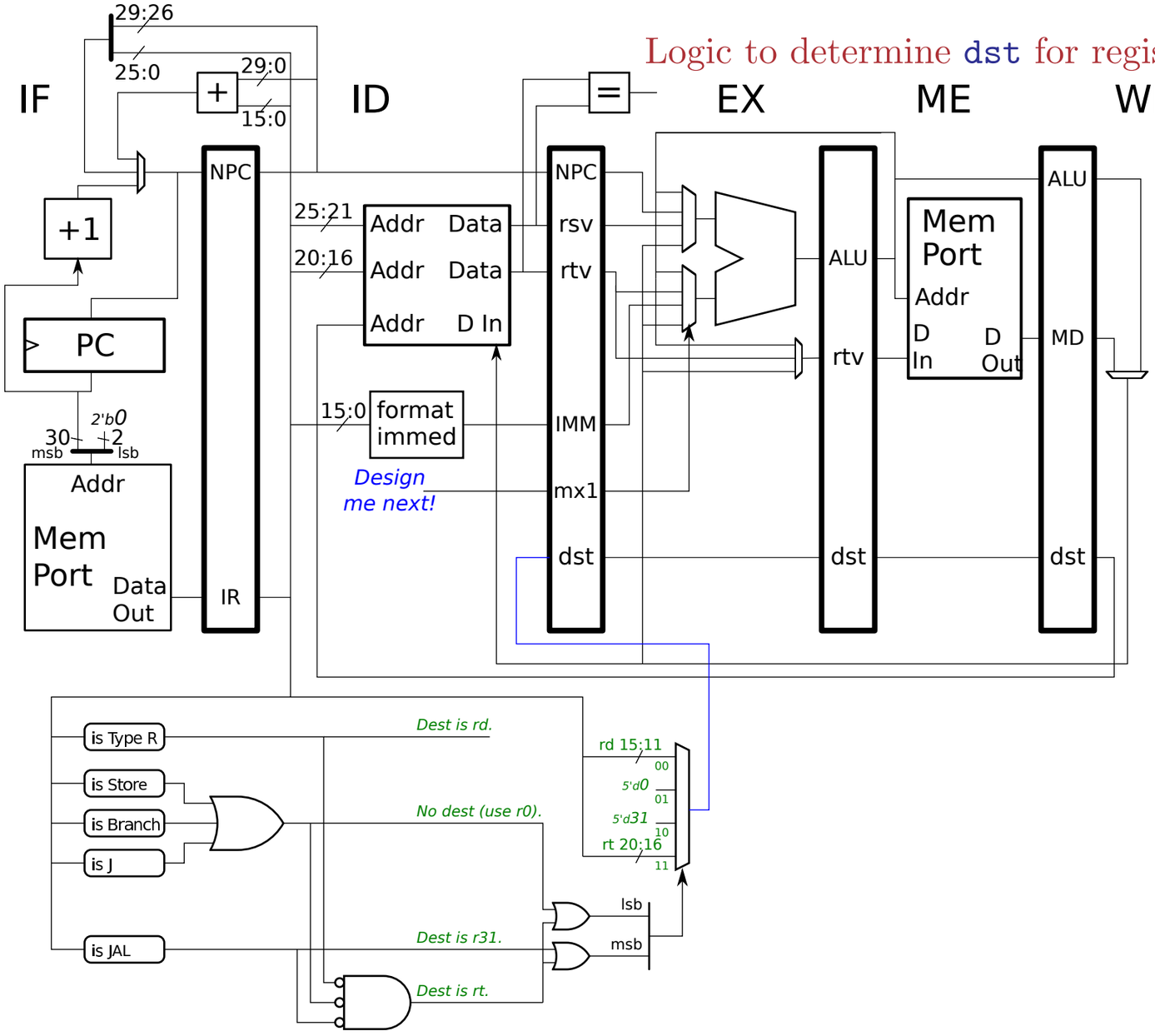


LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-29

Imp-30

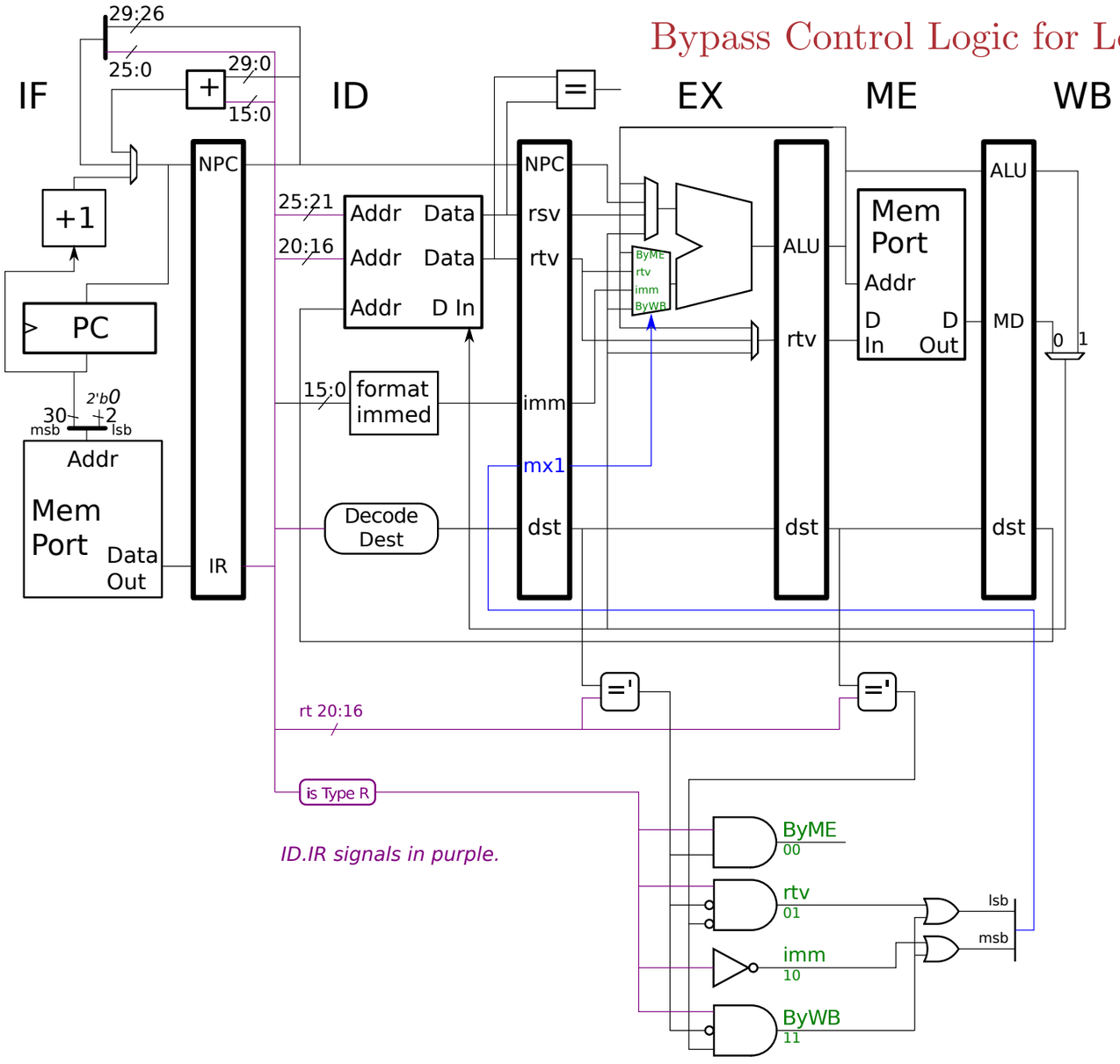
Logic to determine dst for register file.



Imp-30

Bypass Control Logic for Lower ALU Mux

Imp-31



ID.IR signals in purple.

Imp-31

Bypass Control Logic

Imp-32

Control logic not minimized (for clarity).

Control Logic Generating `dst`.

Present in previous implementations, just not shown.

Determines which register gets written based on instruction.

Instruction categories used in boxes such as `= is Store` (some instructions omitted):

`= is Type R`: All Type R instructions.

`= is Store`: All store instructions.

`= is Branch`: branches such as `beq` and `bltz`.

`= is JAL`, `= is J`: Matches the exact instruction.

LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-32

Bypass Control Logic, Continued

Imp-33

Logic Generating `ID/EX.MUX`.

$\boxed{='}$ box determines if two register numbers are equal.

Register number zero is not equal register zero, nor any other register.

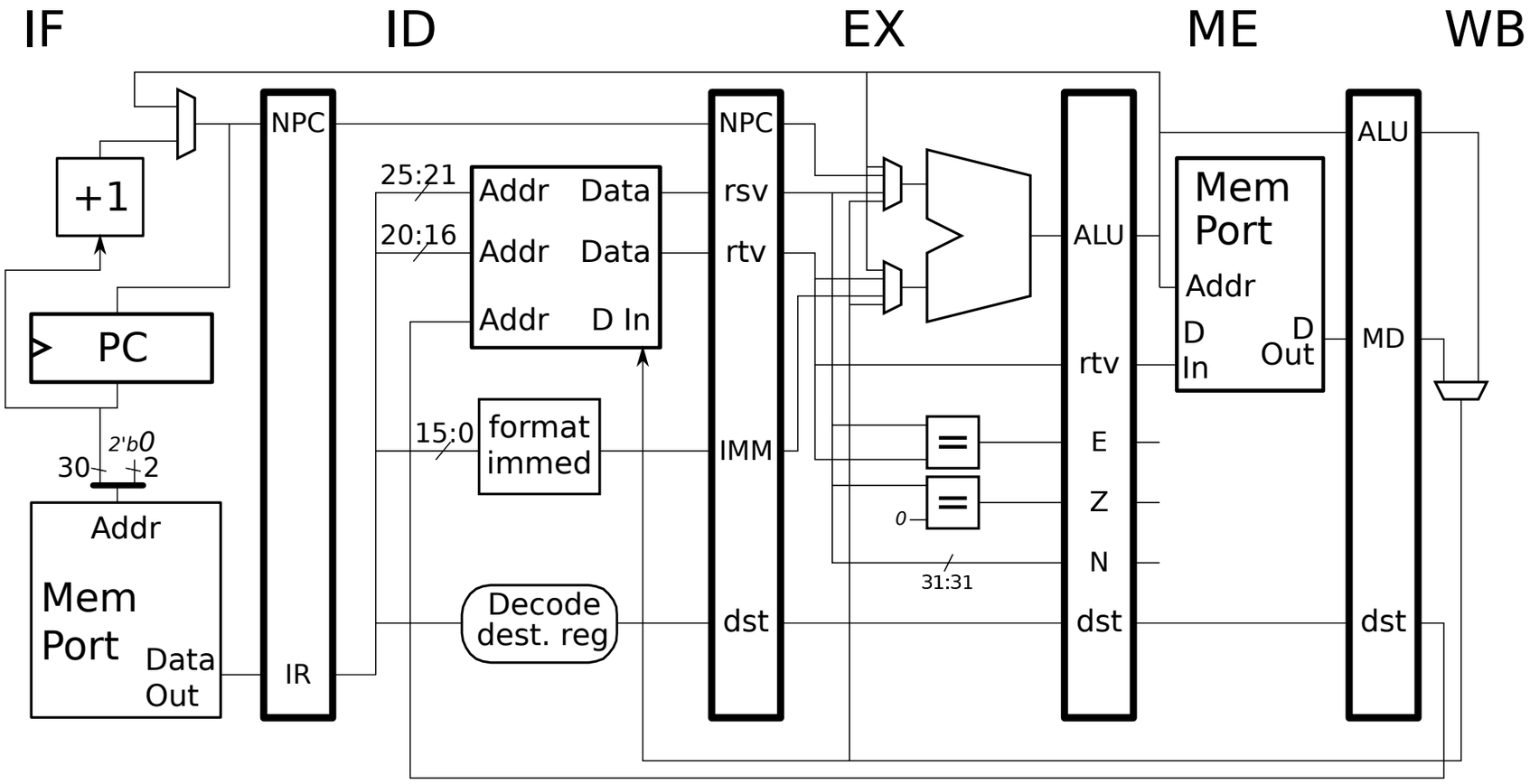
(The bypassed zero value might not be zero.)

Imp-33

Branch Execution

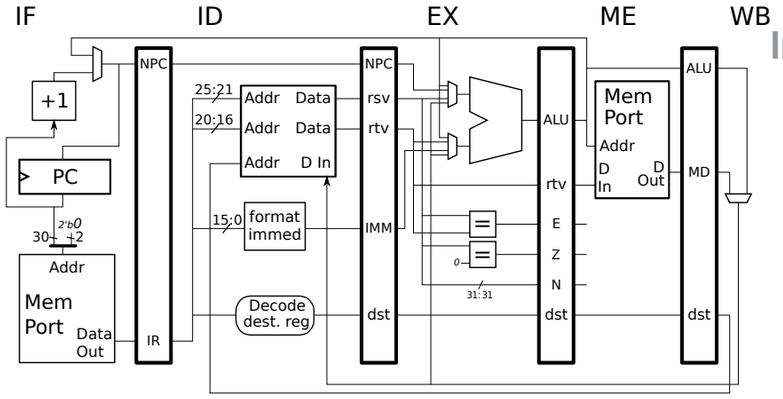
Imp-34

Consider:



Imp-34

Imp-34



Example of **incorrect execution**

```
#I Adr          Cycle   0   1   2   3   4   5   6   7   8
0x100  bgtz r4, TARGET  IF  ID  EX  ME  WB
0x104  sub  r4, r2, r5      IF  ID  EX  ME  WB
0x108  sw   0(r2), r1           IF  ID  EX  ME  WB
0x10c  and  r6, r1, r8              IF  ID  EX  ME  WB
0x110  or   r12, r13, r14
...
TARGET: #  TARGET = 0x200
0x200  xor  r9, r4, r11                 IF  ID  EX  ME  WB
```

Branch is taken yet two instructions past delay slot (`sub`) complete execution.

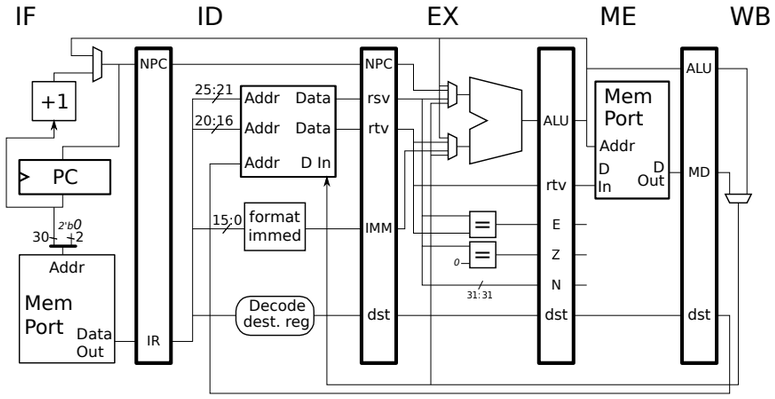
Branch target finally fetched in cycle 4.

Problem: Two instructions following delay slot.

LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-36

Handling Instructions Following a Taken Branch Delay Slot



Option 1: Don't fetch them.

Possible (with pipelining) because ...
... fetch starts (`sw` in cycle 2) ...
... after branch decoded.

(Would be impossible ...
... for non-delayed branch.)

```
#I Adr          Cycle   0   1   2   3   4   5   6   7   8
0x100  bgtz r4, TARGET  IF  ID  EX  ME  WB
0x104  sub  r4, r2, r5      IF  ID  EX  ME  WB
0x108  sw   0(r2), r1           IF  ID  EX  ME  WB
0x10c  and  r6, r1, r8              IF  ID  EX  ME  WB
0x110  or   r12, r13, r14
...
TARGET: #  TARGET = 0x200
0x200  xor  r9, r4, r11                 IF  ID  EX  ME  WB
```


LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.
Imp-36

Imp-37

Handling Instructions Following a Taken Branch

Option 2: Fetch them, but squash (stop) them in a later stage.

This will work if instructions squashed ...
... *before* modifying architecturally visible storage (registers and memory).

Memory modified in ME stage and registers modified in WB stage ...
... so instructions must be stopped before beginning of ME stage.

Can we do it? Depends depends where branch instruction is.

In example, need to squash `sw` before cycle 5.

During cycle 3 `bgtz` in ME ...
... it has been decoded and the branch condition is available ...
... so we know whether the branch is taken ...
... so `sw` can easily be squashed before cycle 5.

Option 2 will be used.

LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.
Imp-37

Instruction Squashing

Imp-38

In-Flight Instruction::
An instruction in the execution pipeline.

Later in the semester a more specific definition will be used.

Squashing:: [an instruction]
preventing an in-flight instruction . . .
. . . from writing registers, memory or any other visible storage.

Squashing also called: *nulling*, *abandoning*, and *cancelling*..

Like an insect, a squashed instruction is still there (in most cases) but can do no harm.

LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-38

Squashing Instruction in Example MIPS Implementation

Imp-39

Two ways to squash.

- Prevent it from writing architecturally visible storage.

Replace destination register control bits with zero. (Writing zero doesn't change anything.)

Set memory control bits (not shown so far) for no operation.

- Change Operation to `nop`.

Would require changing many control bits.

Squashing shown that way here for brevity.

Illustrated by placing a `nop` in `IR`.

Why not replace squashed instructions with target instructions?

Because there is no straightforward and inexpensive way . . .
. . . to get the instructions *where and when* they are needed.

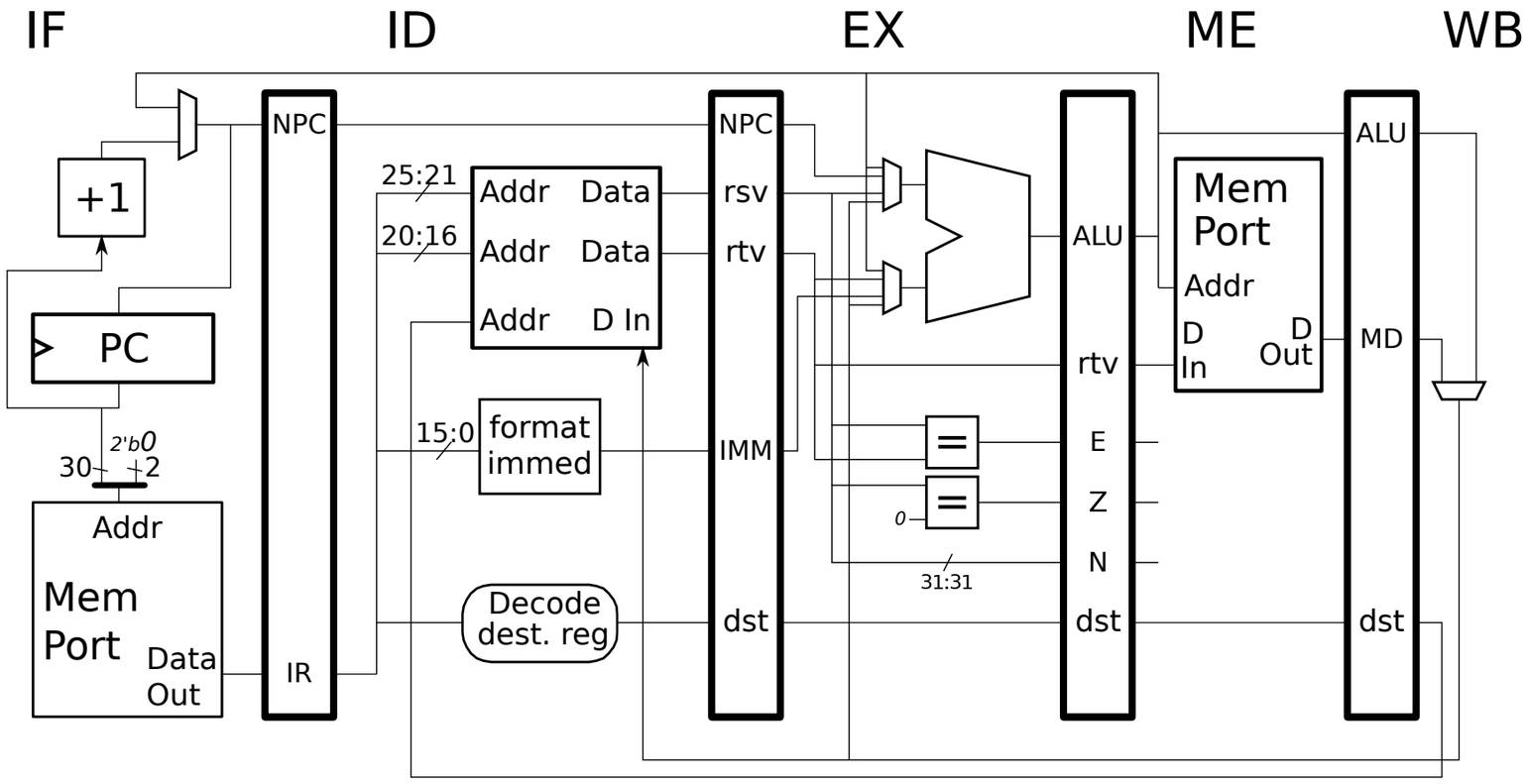
(Curvysideways and expensive techniques covered in Chapter 4.)

LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-39

Imp-40

MIPS implementation used so far.



Imp-40

Imp-41

Example of correct execution

```
#I Adr          Cycle   0   1   2   3   4   5   6   7   8
0x100  bgtz r4, TARGET  IF  ID  EX  ME  WB
0x104  sub  r4, r2, r5      IF  ID  EX  ME  WB
0x108  sw   0(r2), r1           IF  IDx
0x10c  and  r6, r1, r8              IFx
0x110  or   r12, r13, r14
...
TARGET: #  TARGET = 0x200
0x200  xor  r9, r4, r11                 IF  ID  EX  ME  WB
```

Branch outcome known at end of cycle 2 . . .

. . . wait for cycle 3 when doomed instructions (`sw` and `and`) in flight . . .

. . . and squash them so in cycle 4 they act like `nop`s.

Two cycles (2, and 3), are lost.

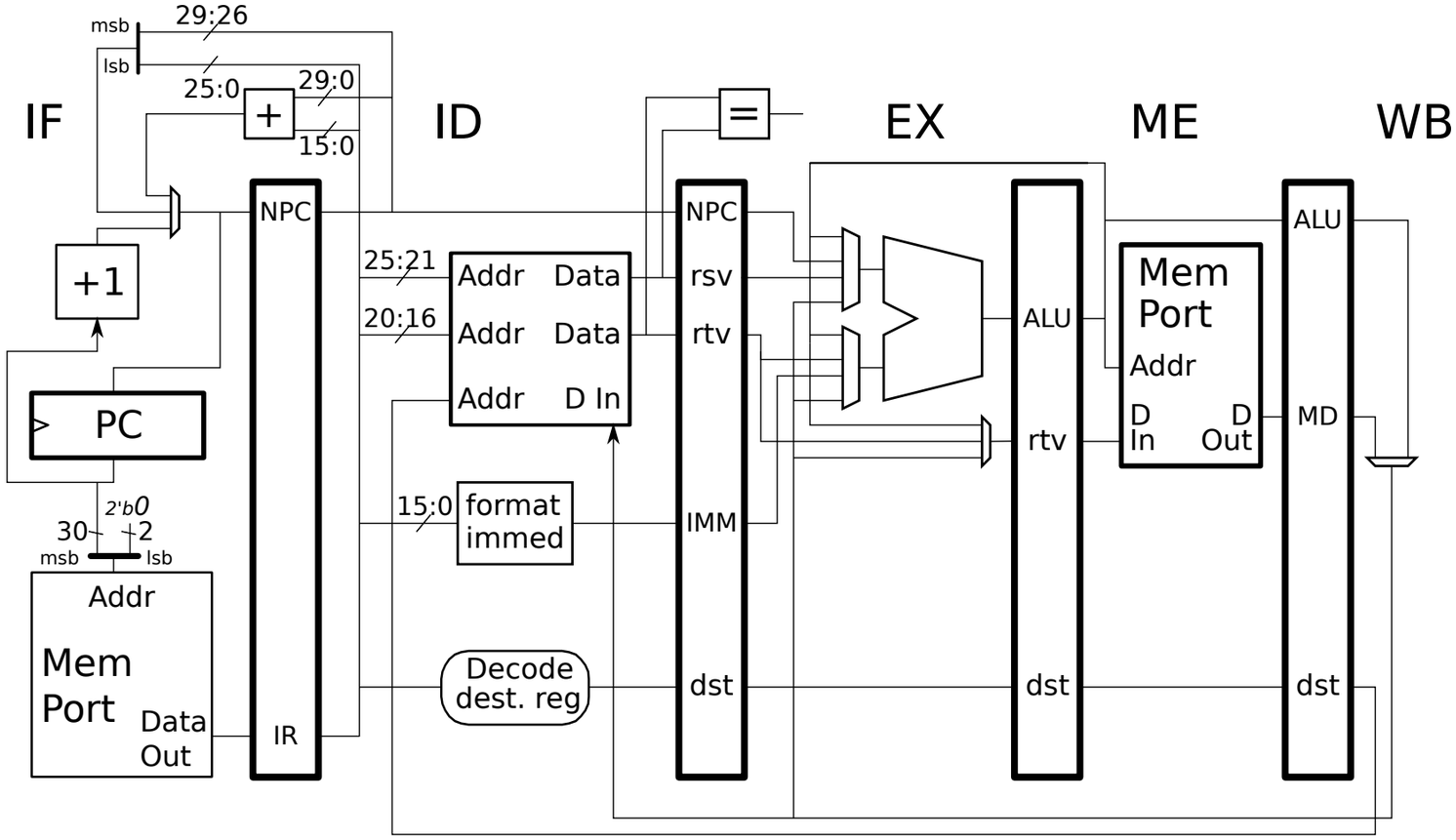
The two cycles called a *branch penalty*.

Two cycles can be alot of cycles, is there something we can do?

LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.
Imp-41

Yes: Zero-Cycle Branch Delay Implementation

Imp-42



Compute branch target address in ID stage.

LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-42

Zero-Cycle Branch Delay Implementation

Imp-43

Compute branch target and condition in ID stage.

Workable because register values not needed to compute branch address and . . .
. . . branch condition can be computed quickly.

Now how fast will code run?

```
#I Adr          Cycle   0   1   2   3   4   5   6   7   8
0x100  bgtz r4, TARGET  IF  ID  EX  ME  WB
0x104  sub  r4, r2, r5      IF  ID  EX  ME  WB
0x108  sw   0(r2), r1
0x10c  and  r6, r1, r8
0x110  or   r12, r13, r14
...
TARGET: #  TARGET = 0x200
0x200  xor  r9, r4, r11         IF  ID  EX  ME  WB
```

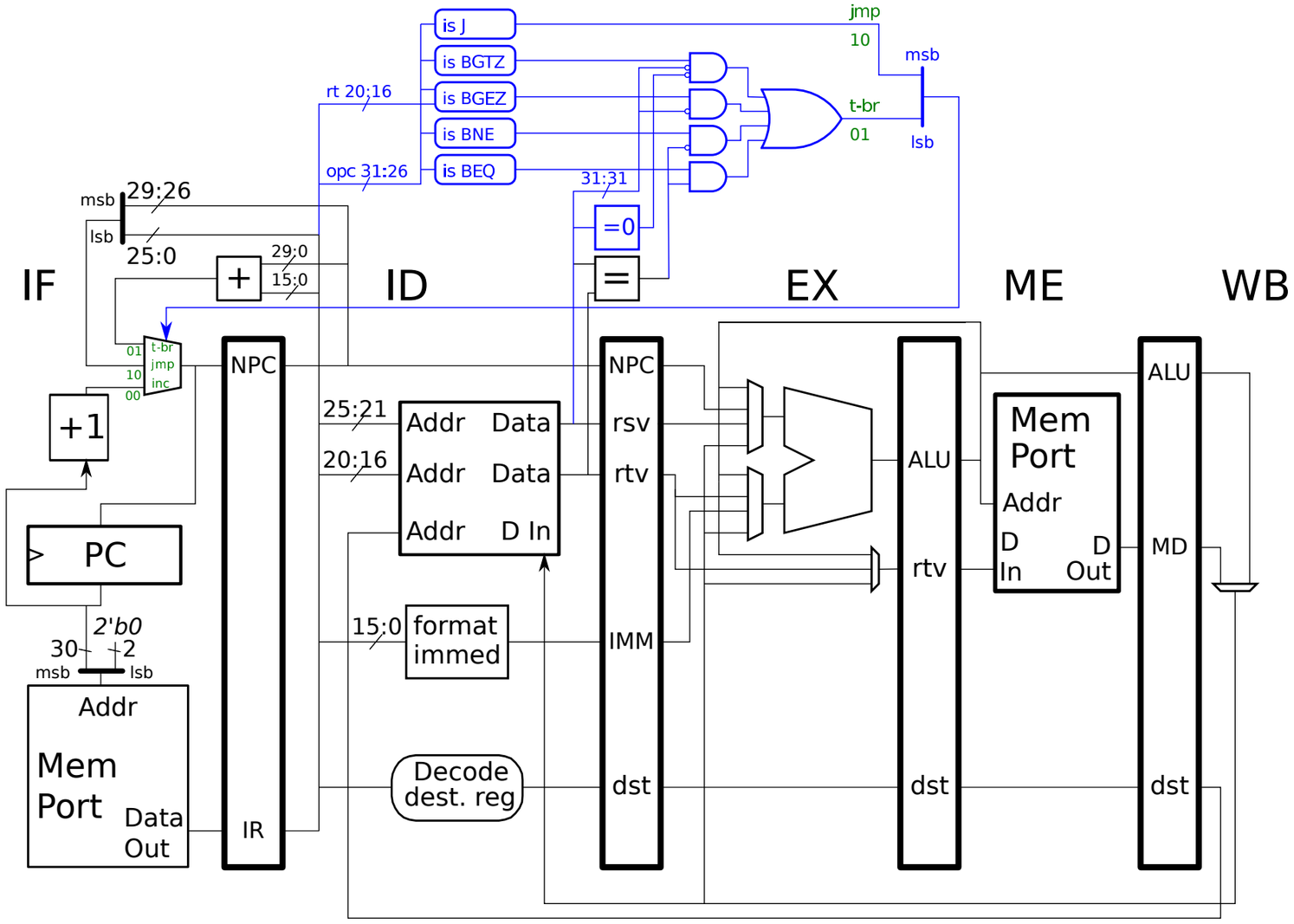
No penalty, not a cycle wasted!!

LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-43

Control Logic for some Control Transfers

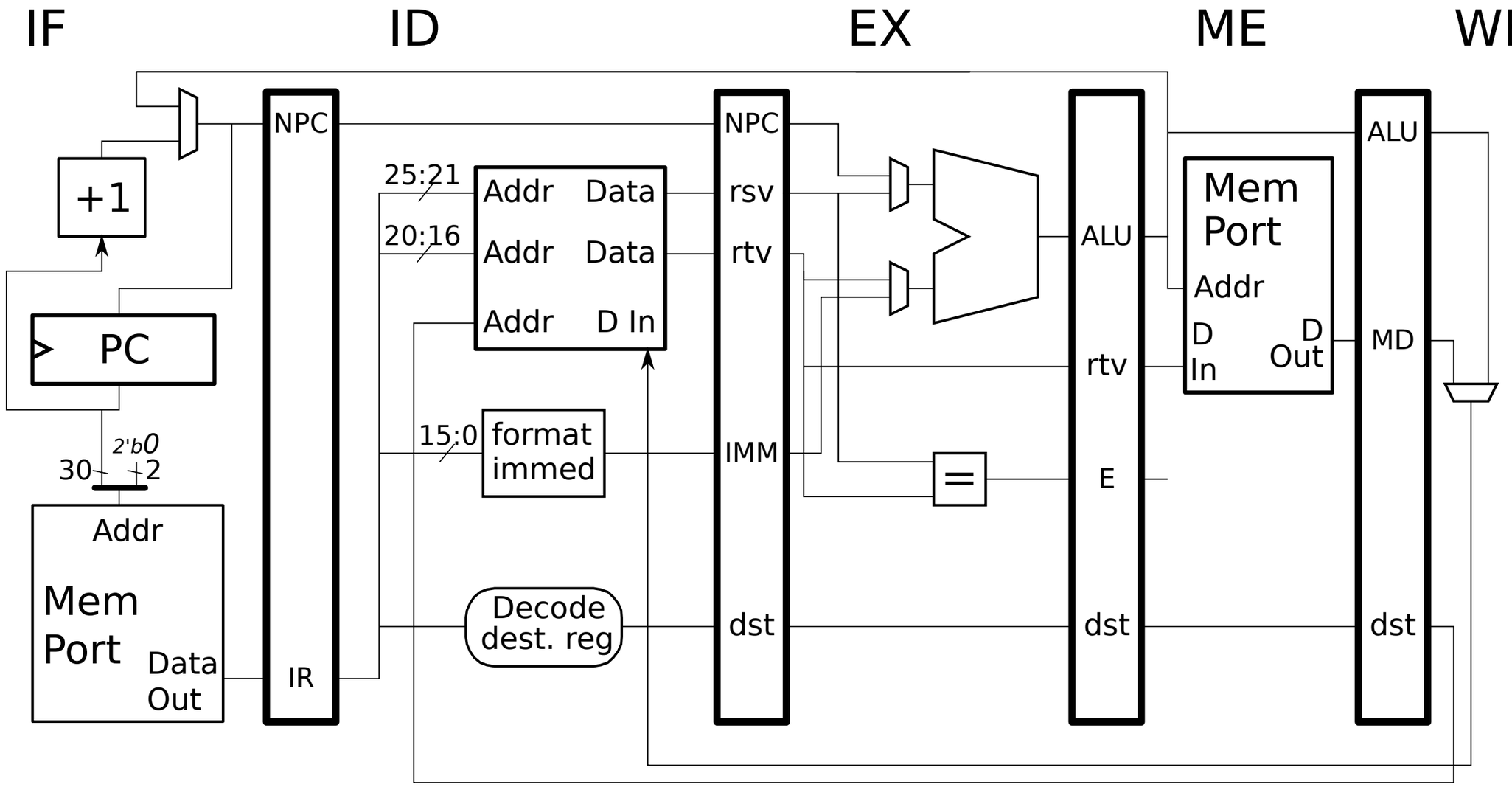
Imp-44



Imp-44

Non-Bypassed MIPS

Imp-45

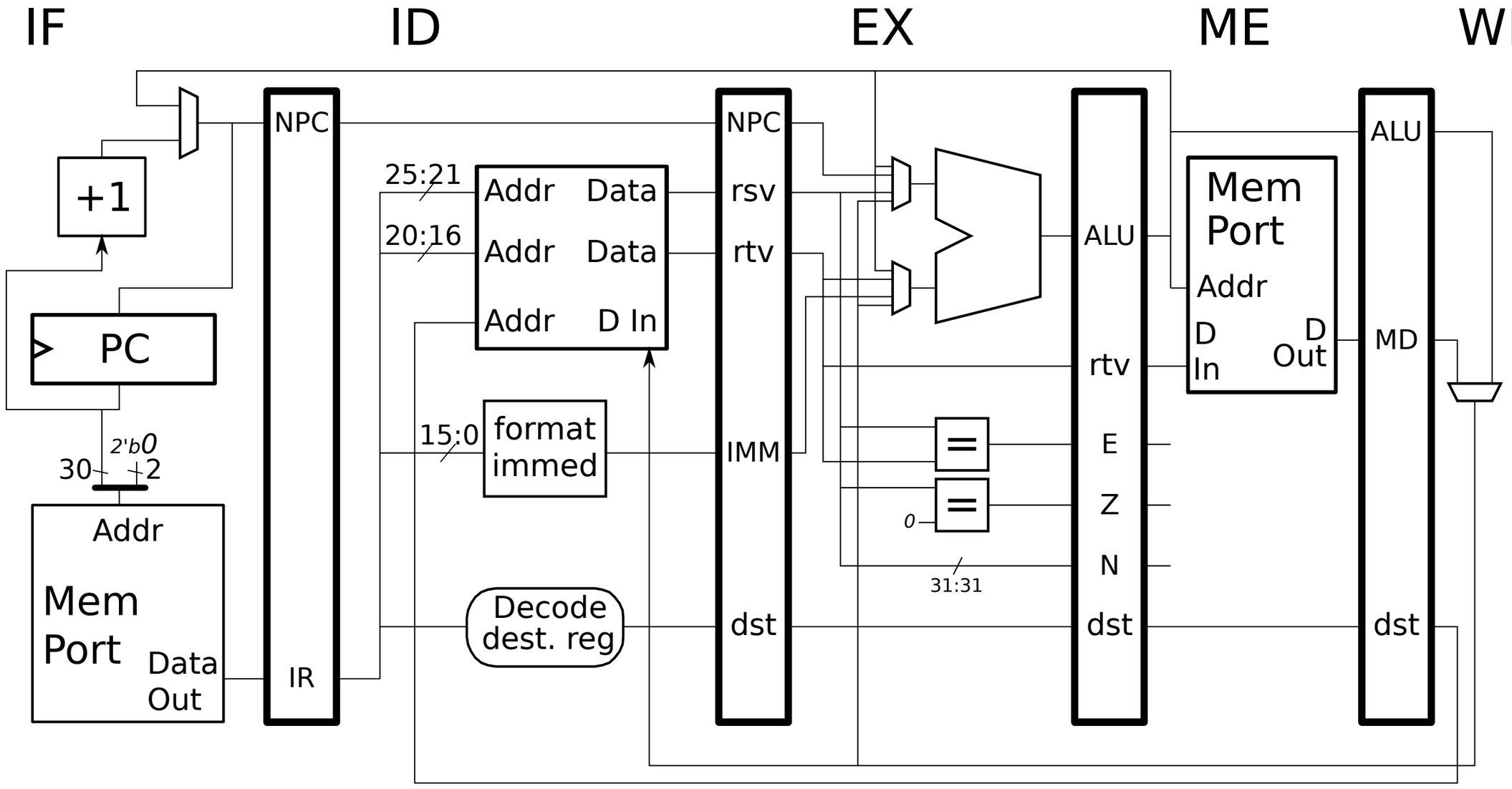


LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-45

Bypassed MIPS

Imp-46

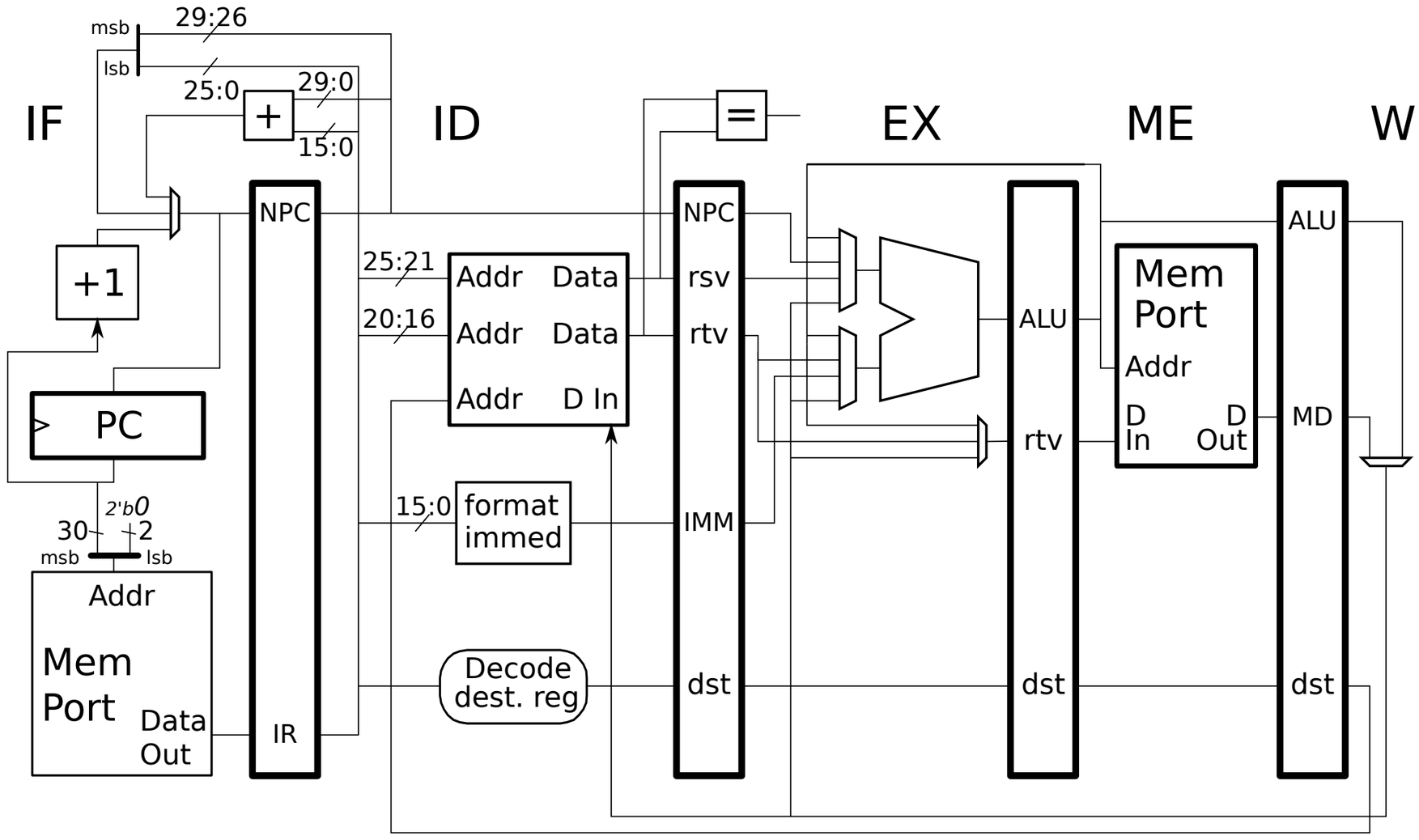


LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-46

ID Branch MIPS

Imp-47



LSU EE 4720 Lecture Transparency. Formatted 12:37, 6 February 2019 from lsli06-TeXize.

Imp-47