Georgia Tech Computer Science

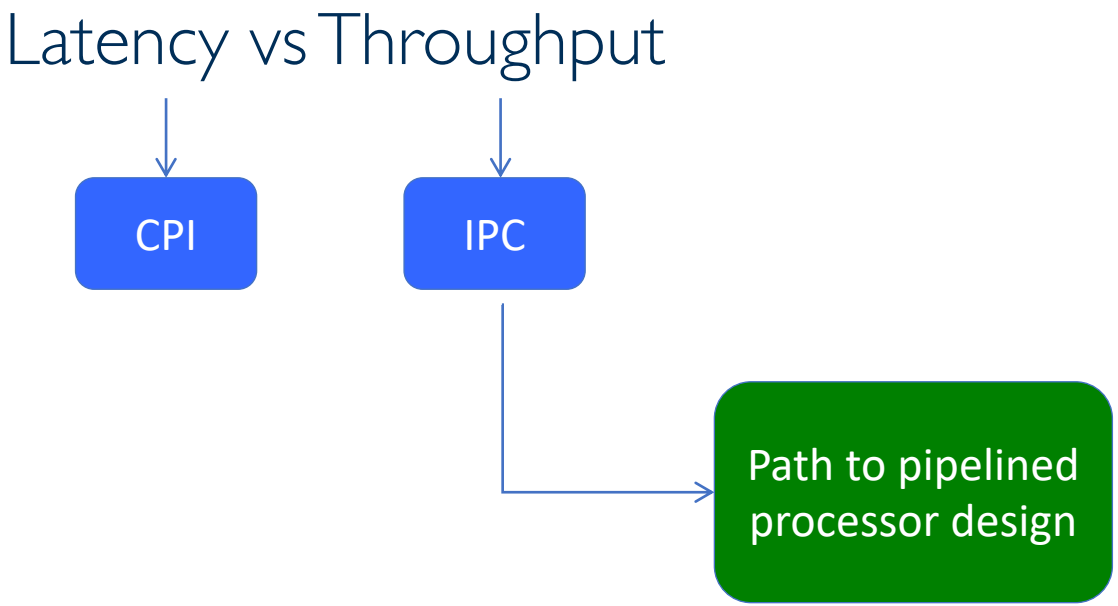
CS2200
Systems and Networks
Spring 2022

Lecture 10: Pipelining

Alexandros (Alex) Daglis
School of Computer Science
Georgia Institute of Technology
adaglis@gatech.edu

Lecture slides adapted from Bill Leahy and Charles Lively of Georgia Tech

Going faster…



Bill's Sandwich Shop

station I
(place order)

station II
(select bread)

station III
(cheese)

station IV
(meat)

station V
(veggies)

Bill's Sandwich Shop

station I	station II	station III	station IV	station V
(place order)	(select bread)	(cheese)	(meat)	(veggies)

Bill's Sandwich Shop

station I	station II	station III	station IV	station V
(place order)	(select bread)	(cheese)	(meat)	(veggies)

Bill's Sandwich Shop

station I	station II	station III	station IV	station V
(place order)	(select bread)	(cheese)	(meat)	(veggies)

Bill's Sandwich Shop

station I	station II	station III	station IV	station V
(place order)	(select bread)	(cheese)	(meat)	(veggies)

Net result: One complete sandwich every five cycles.

Bill's Mega-Sandwich Shop





station I
(place order)

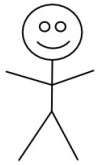
station II
(select bread)

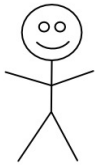
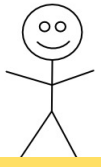
station III
(cheese)

station IV
(meat)

station V
(veggies)

"What will You Have?"

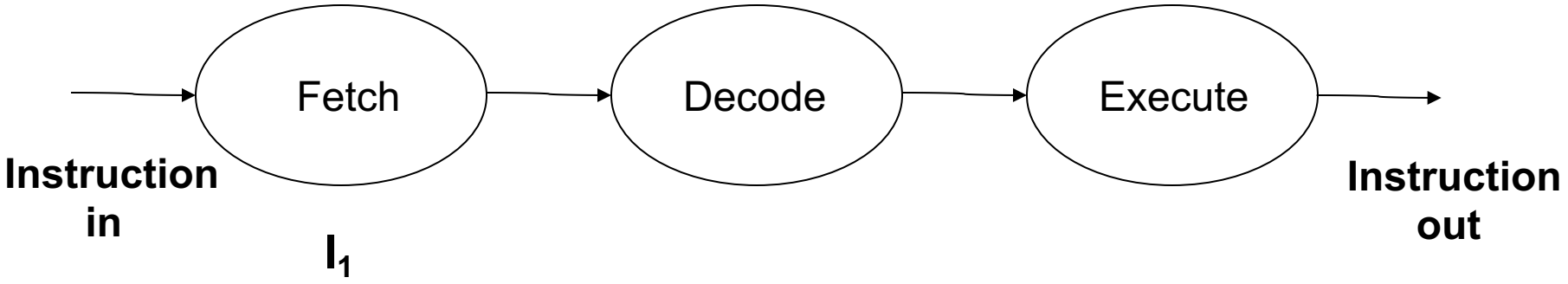



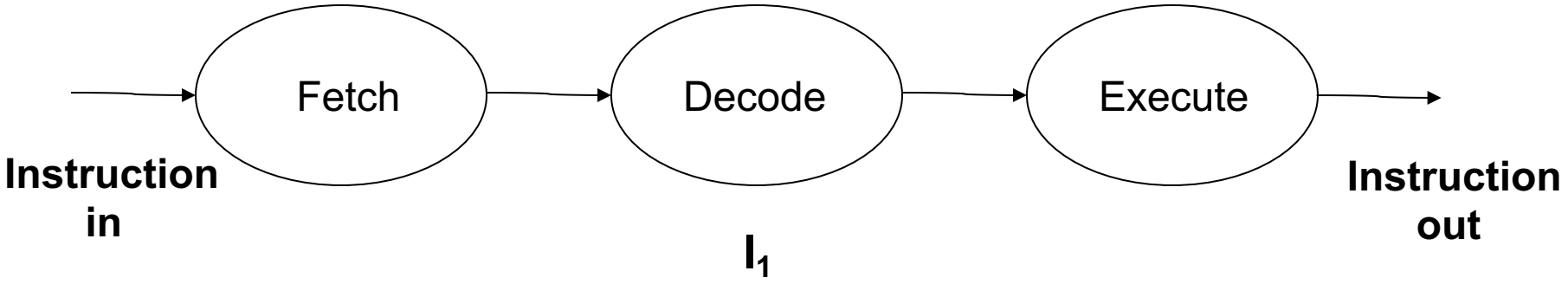
Manager

Net result: One complete sandwich every cycle! (Once you fill the pipeline.)
A 5x speedup!

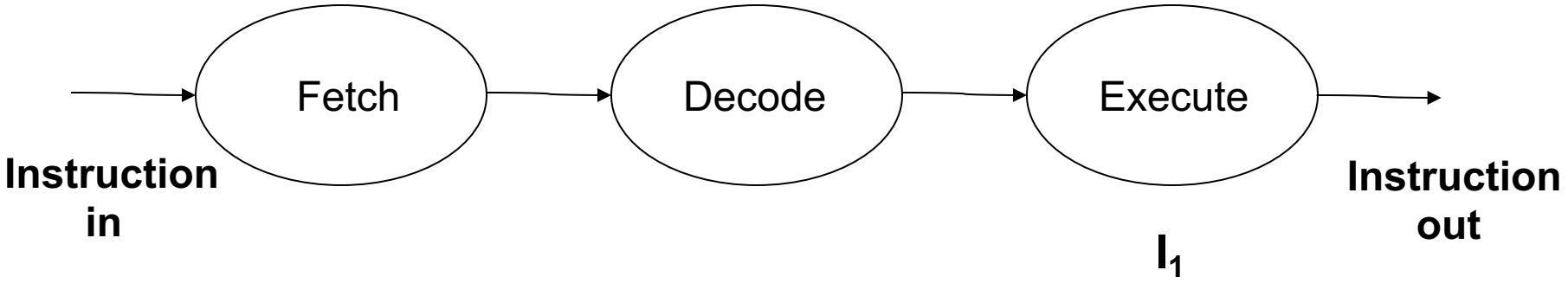
Interpreting an instruction



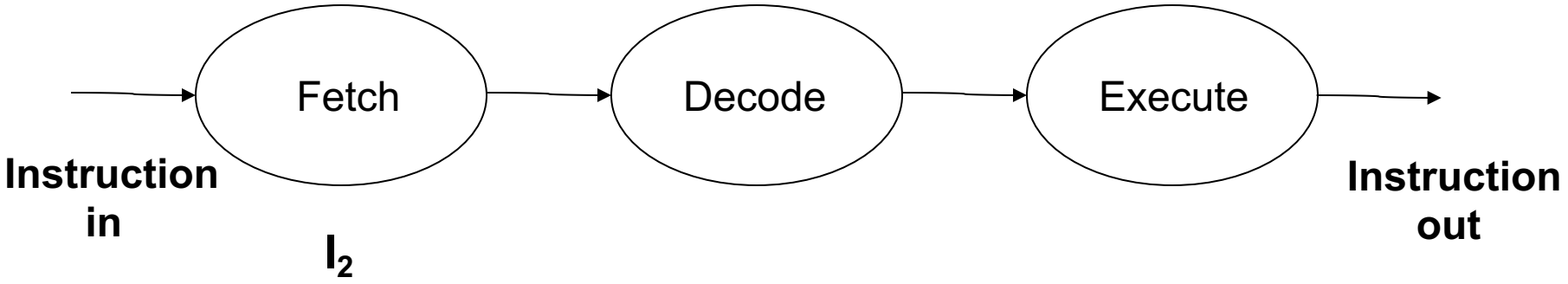
Interpreting an instruction



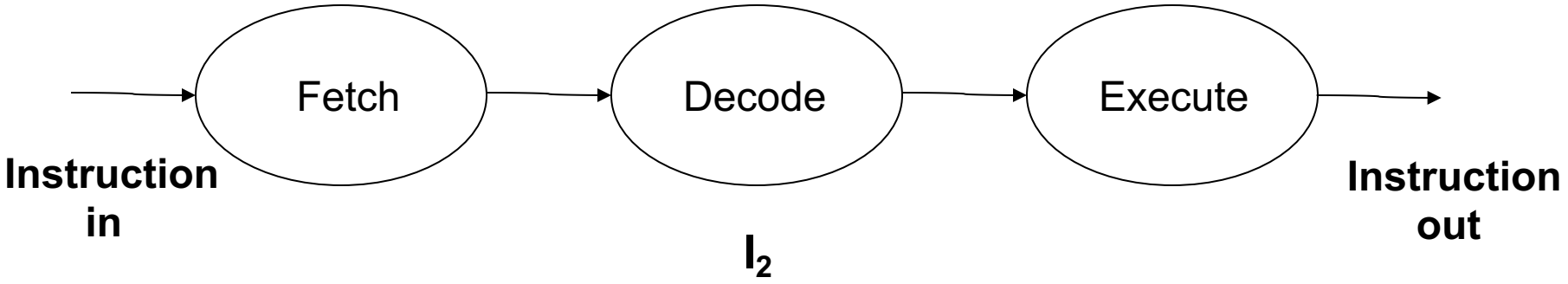
Interpreting an instruction



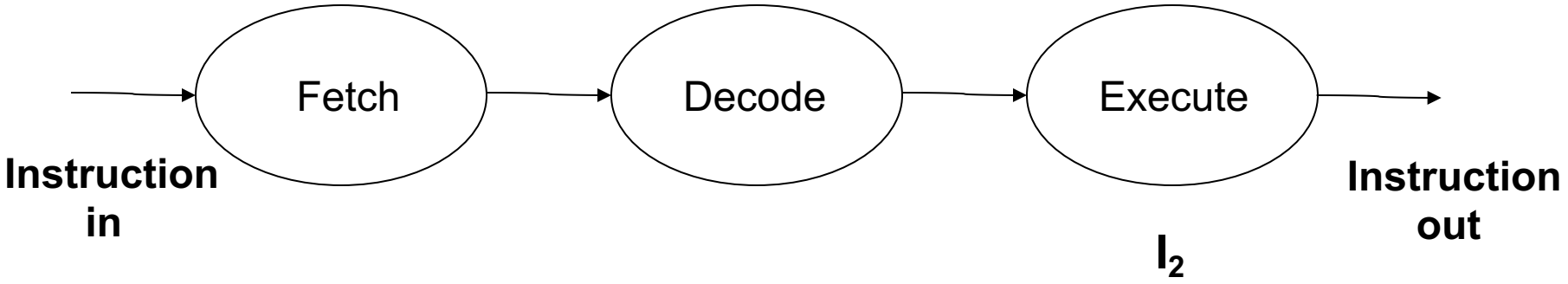
Interpreting an instruction



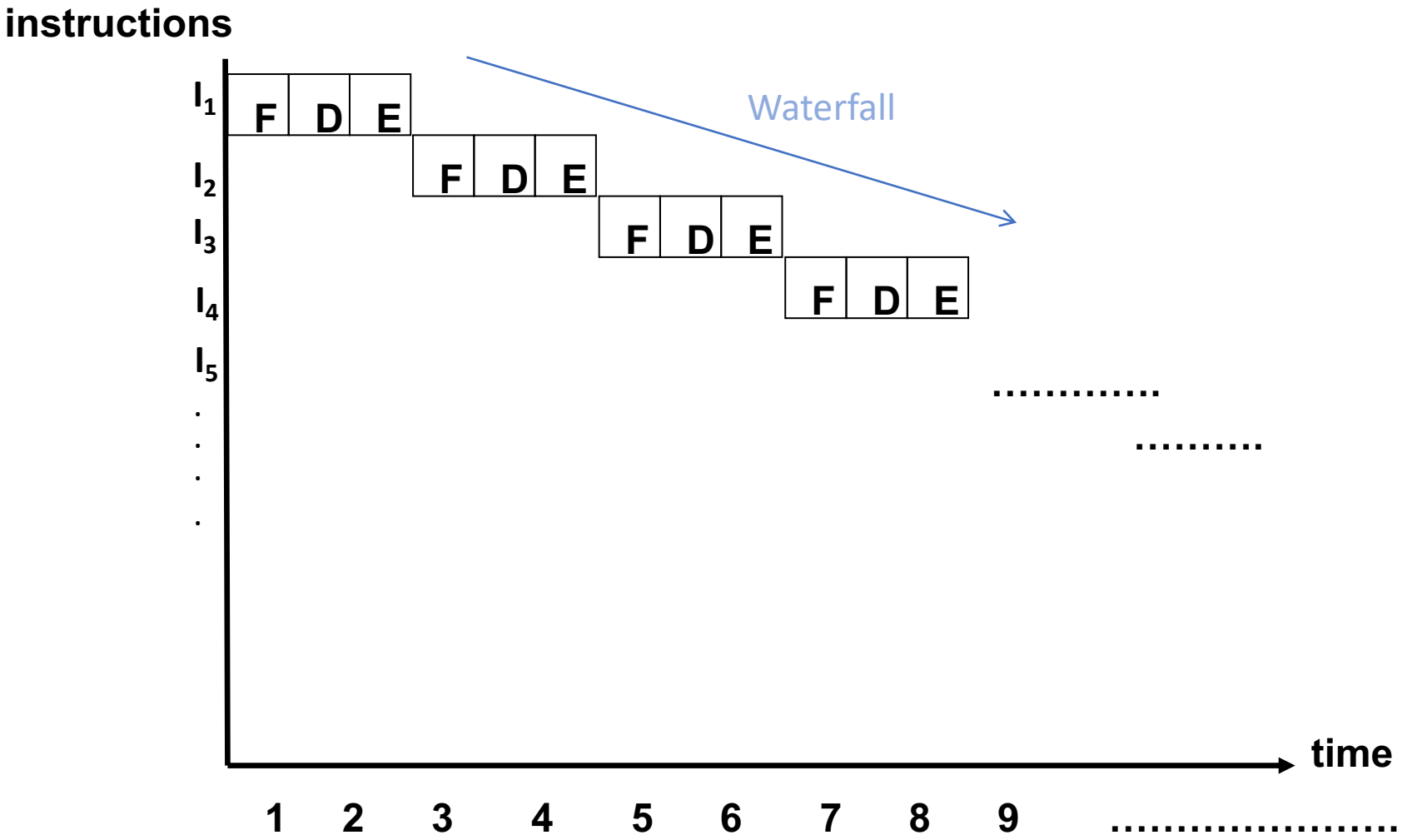
Interpreting an instruction



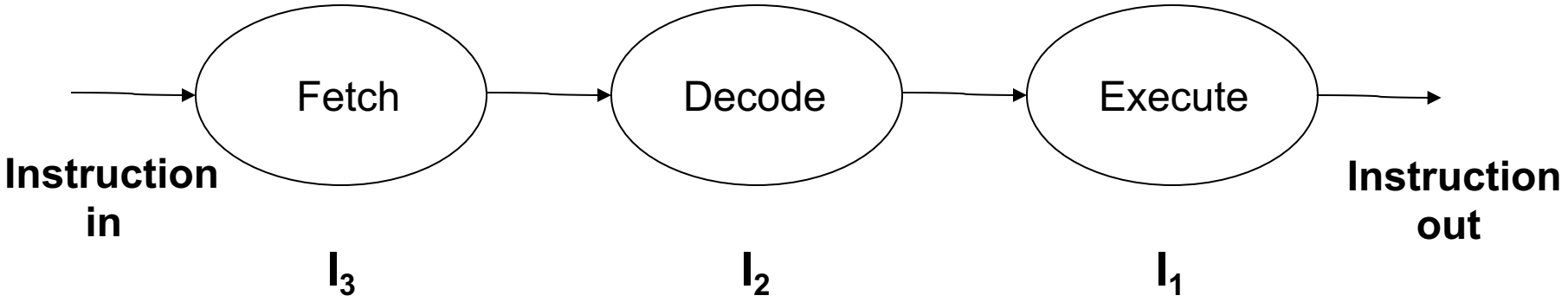
Interpreting an instruction



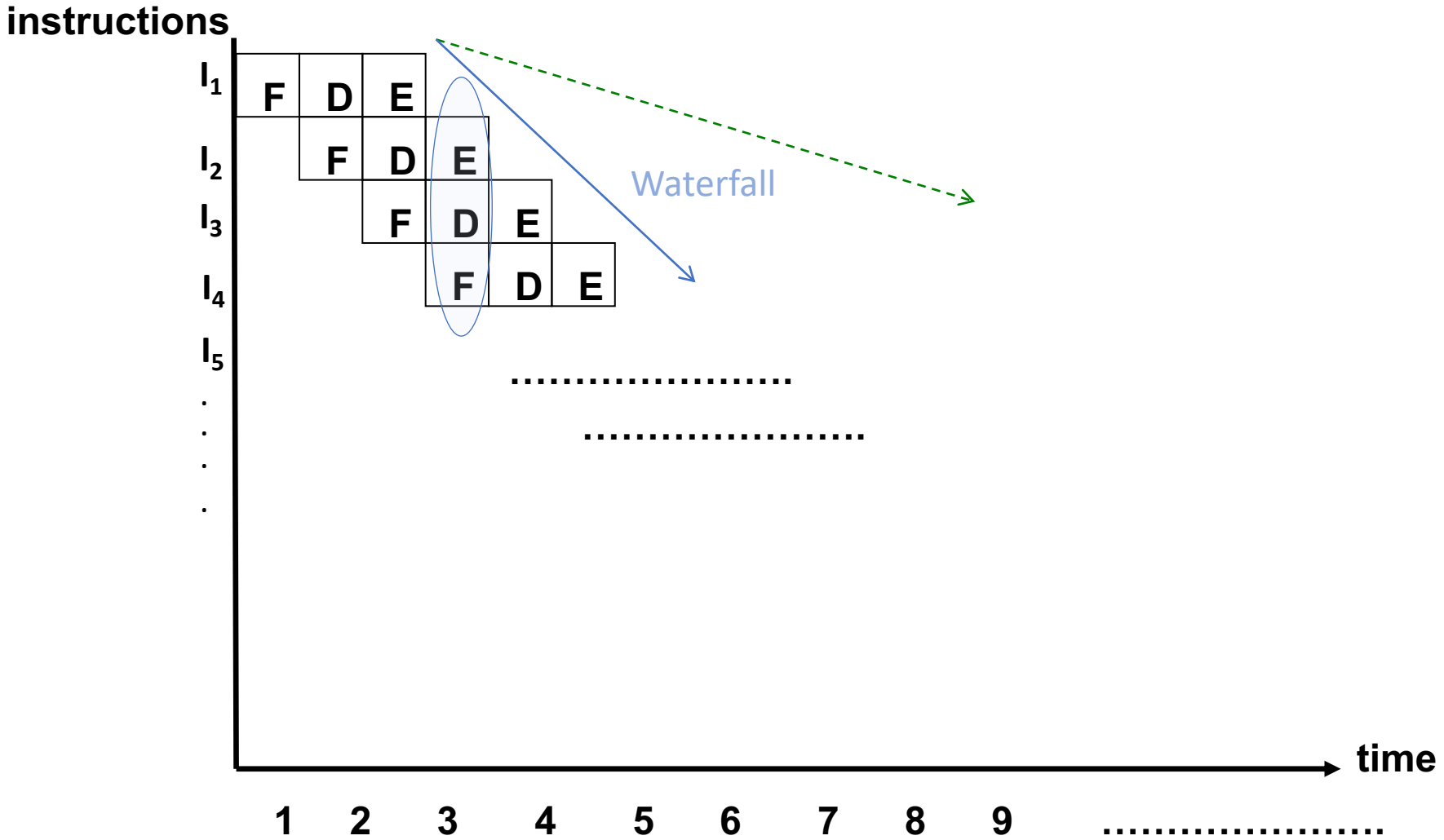
One at a time…



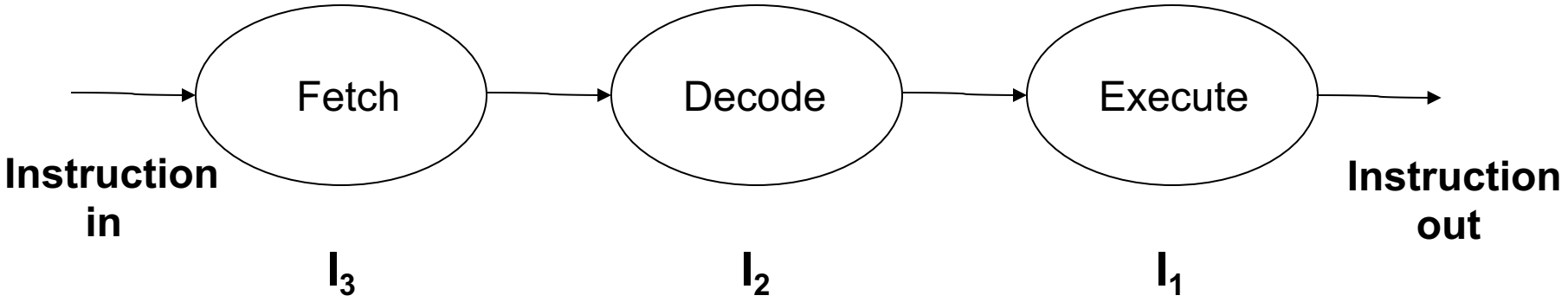
Three at a time?



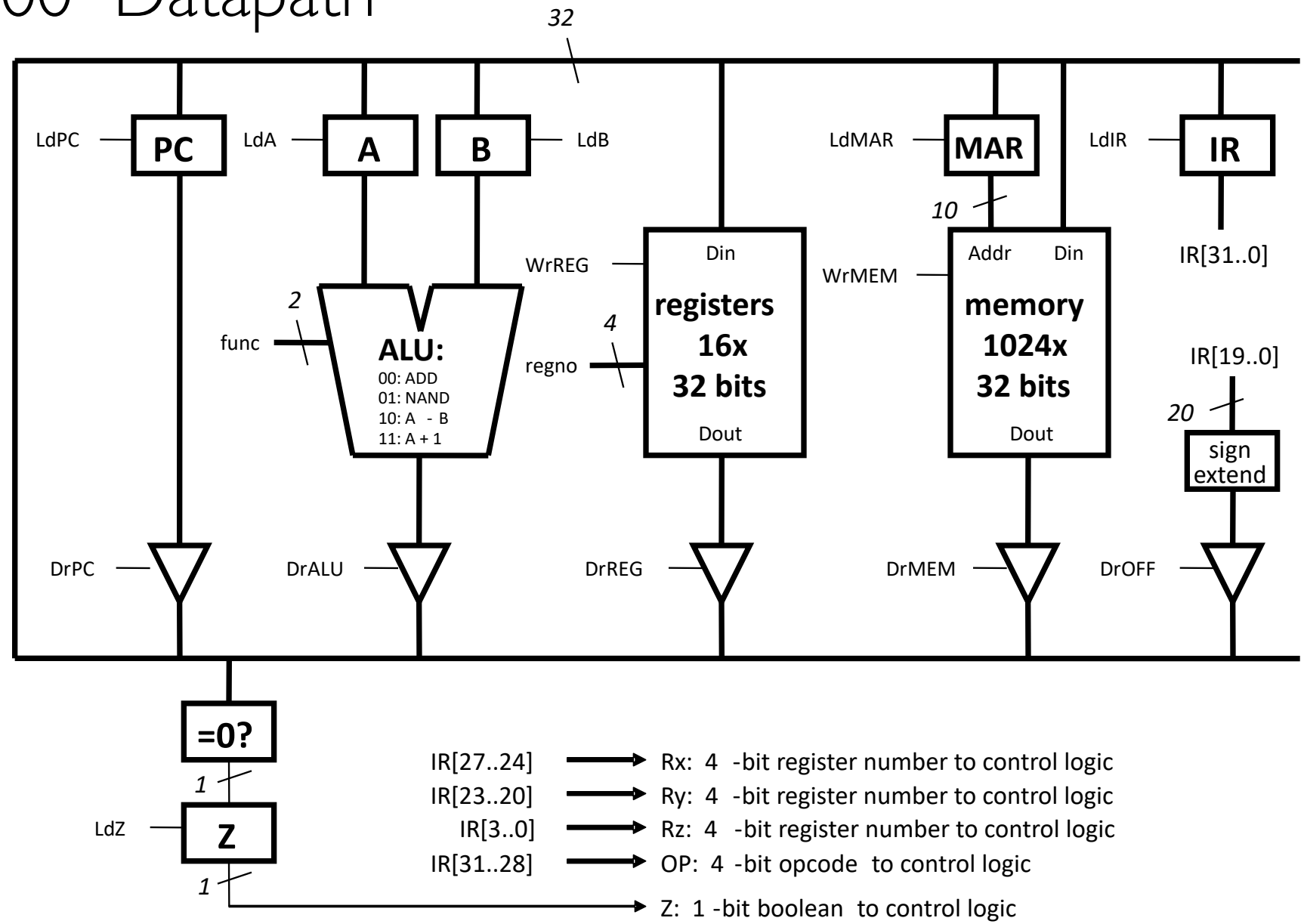
Overlapping execution



What do we need in the datapath?

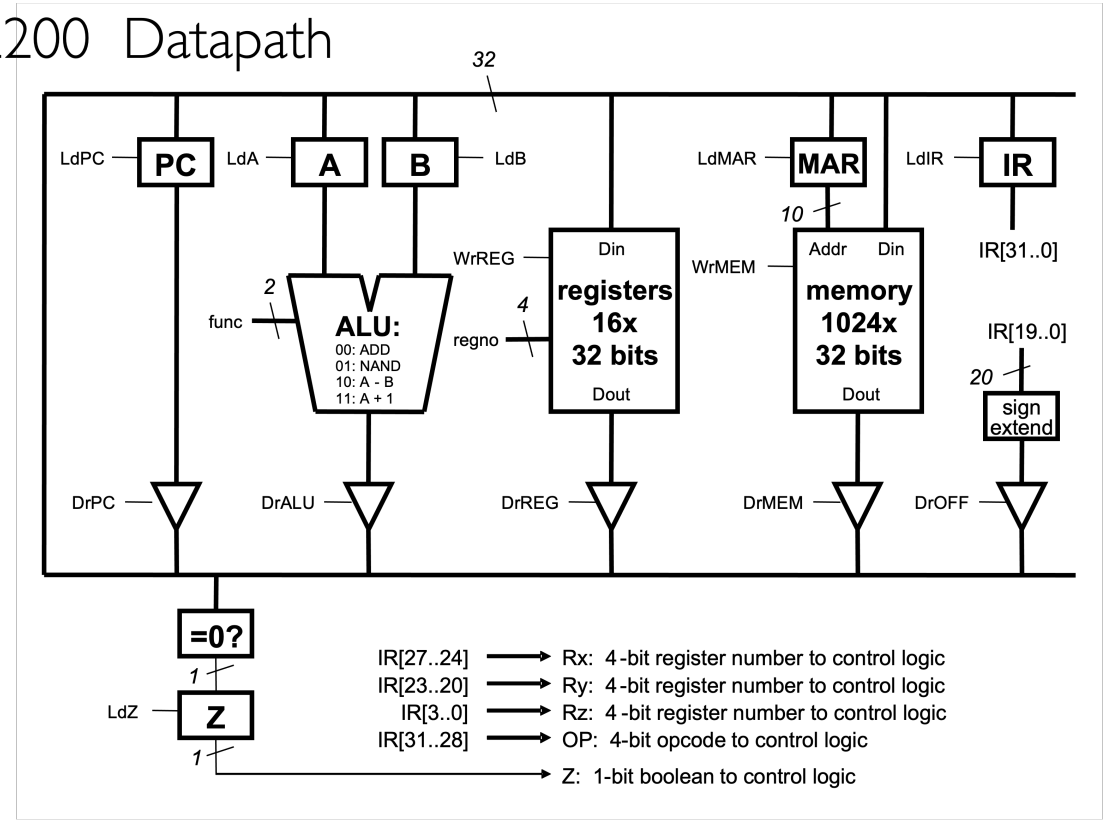


LC-2200 Datapath



What units are used?

LC-2200 Datapath



Macro State	Units in Use
FETCH	IR, ALU, PC, MEM
DECODE	IR
EXECUTE (ADD)	IR, ALU, Reg-file
EXECUTE (LW)	IR, ALU, Reg-file, MEM, Sign extender

Imitation: The Sincerest Form of Flattery

station I
(place order)
New (5th order)

station II
(select bread)
4th order

station III
(cheese)
3rd order

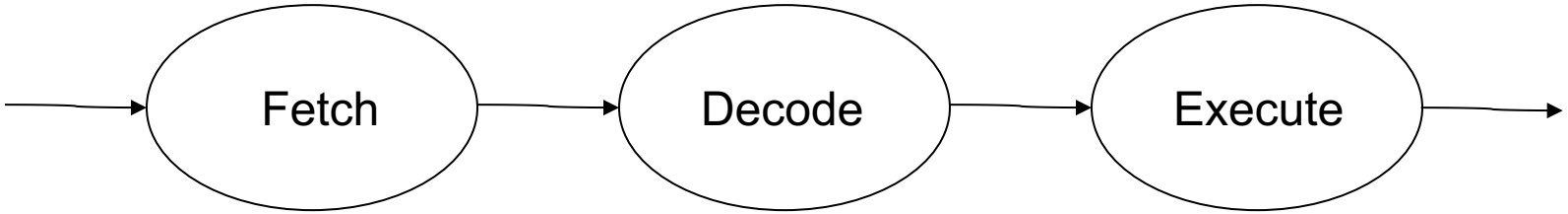
station IV
(meat)
2nd order

station V
(veggies)
1st order

Points to note:

- Work done by each station is roughly equal
- Every sandwich goes through every station, regardless of what you want or don't want
- Order form and partially assembled sandwich is passed from one station to the next
- Each station does part of the work for assembling a sandwich

How do we mimic sandwich assembly?



- Unequal division of labor
- Violates first principle in sandwich assembly line

Bill's Cost-Cutting Measure

station I
(place order)
New (5th order)

station II
(select bread)
4th order

station III
(cheese)
3rd order

station IV
(meat)
2nd order

station V
(veggies)
1st order

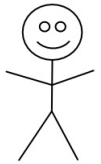
Bill's Cost-Cutting Measure



Station 1
(place order)
3rd order

station II
(select bread)
2nd order

station III
(cheese, meat, veggies)
1st order

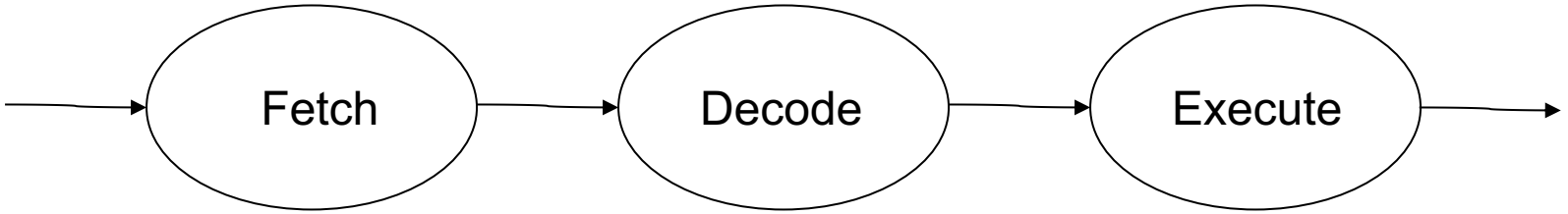

Disaster!

- Increased waiting times
- One sandwich every 3 cycles, not one every cycle!
- Two workers who only work about a third of the time

Bill originally used 5 stages rather than 3 in his sandwich pipeline because…

A. He copied the instruction pipeline
B. He could only find 5 workers
C. He is solving the unemployment problem
D. He wants to make sure that the amount of work per stage is roughly the same
E. He is a nice guy
F. You are making us hungry...

How do we mimic sandwich assembly?



- Get to the basics!
- What needs to happen for every instruction?
  1. Fetch into IR and increment PC
  2. Decode instruction and read register contents
  3. Perform arithmetic/logic (maybe)
  3. Perform address computation (maybe)
  4. Fetch/store memory operand (maybe)
  5. Write to register (maybe)

Each step is roughly the same amount of work

How can we know what registers to read?

- What does the "bread guy" care about?
- How does the "bread guy" know what bread?
- It's the order form!

- What's the equivalent of the order form in the instruction pipeline?
- IR

LC-2200 Instruction set

- R-type instructions (add, nand):

bits 31-28: opcode; **bits 27-24: reg X;**
bits 23-20: reg Y; bits 19-4: unused (should be all 0s);
bits 3-0: reg Z

- I-type instructions (addi, lw, sw, beq):

bits 31-28: opcode; **bits 27-24: reg X**
bits 23-20: **reg Y**; bits 19-0: Imm. Offset

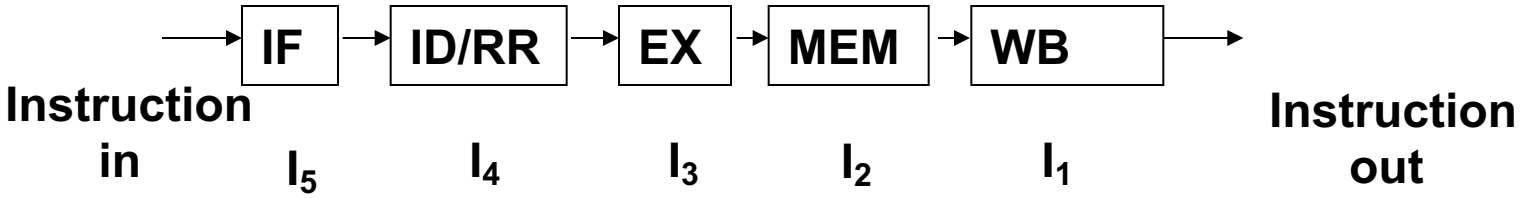
- J-type instructions (jalr):

bits 31-28: opcode; **bits 27-24: reg X**
bits 23-20: reg Y; bits 19-0: unused

- O-type instructions (halt):

bits 31-28: opcode; bits 27-0: unused

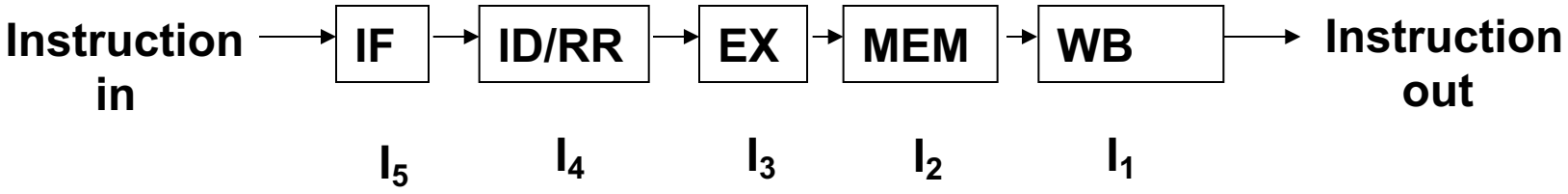
Every instruction goes through



- Some stages don't do anything for some instructions
- Each stage works on a different instruction

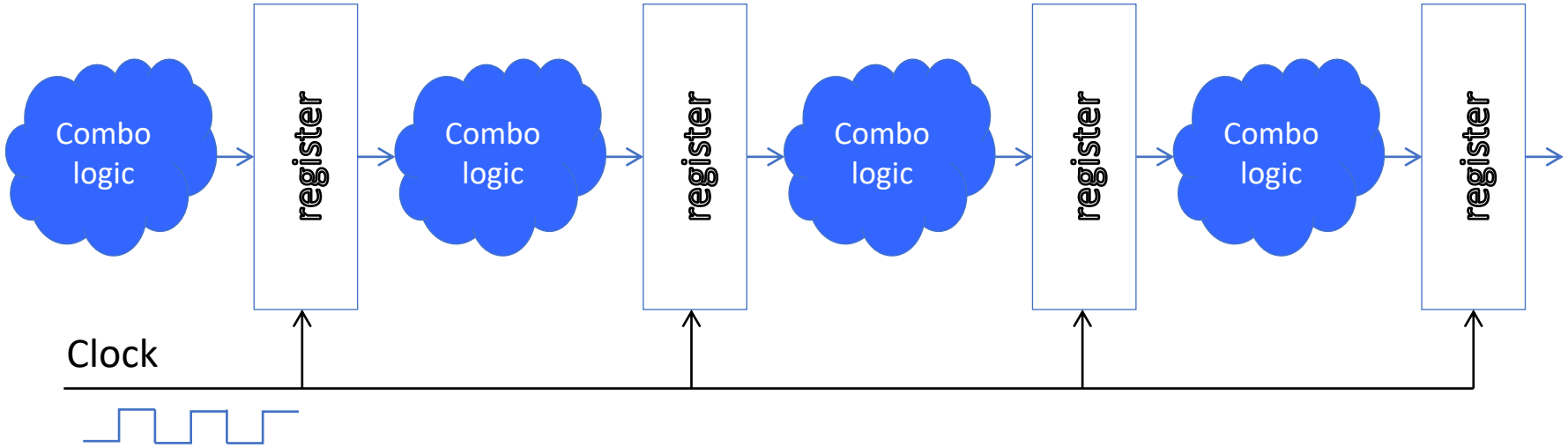
IF - fetch instruction into IR and increment PC
ID/RR - decode and read register contents
EX - perform arithmetic/logic (maybe)
- perform address computation (maybe)
MEM - fetch/store memory operand (maybe)
WB - write to register (maybe)

What moves between stages?



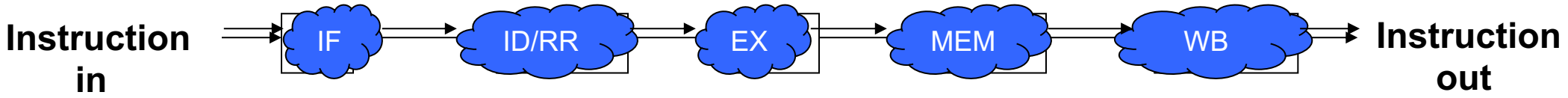
- Think of our sandwich assembly line
- Each station passes two things to the next
  - The order form
  - A partially assembled sandwich
- Analogies with our data path
  - Is the IR equivalent to the order form?
  - What is a "partially assembled data sandwich"?

Electrical isolation between stages



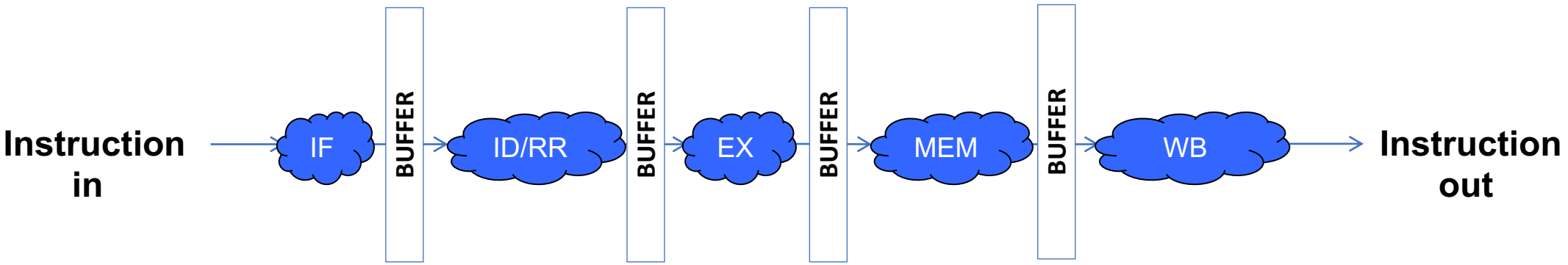
- Registers act as a wall to isolate the actions of each combinational logic cloud!
- On each clock tick, the registers send their output to drive the next stage of combinational logic

What's in these pipeline boxes?



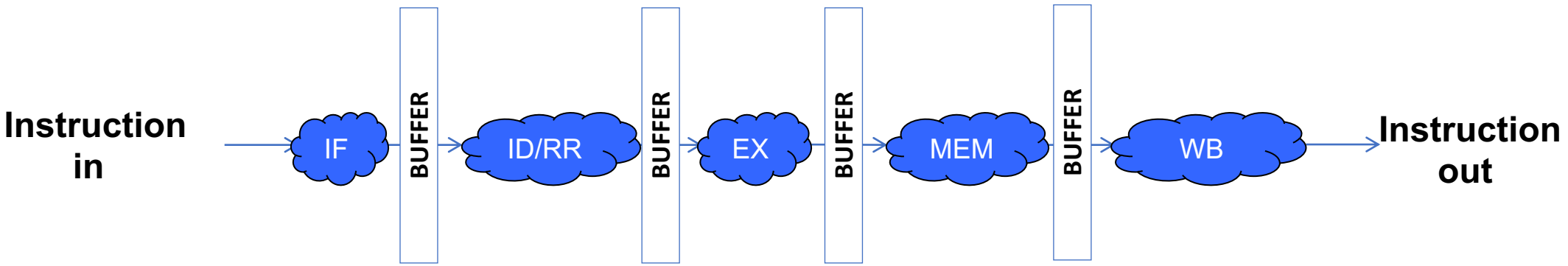
- How do we separate our pipeline boxes?
- The boxes are really just combinational logic clouds
- So we already know how to separate them…

This is where buffers come from…



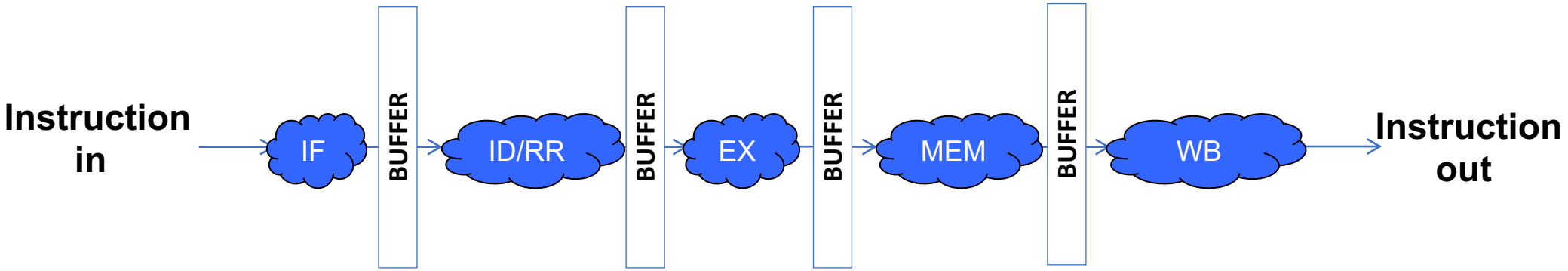
- There is combinational logic in each cloud for performing the actions of that stage
- The buffers are merely clocked registers for passing the partially assembled results

Again: What does each stage do?



IF - fetch instruction into IR and increment PC
ID/RR - read register contents
EX - perform arithmetic/logic (maybe)
- perform address computation (maybe)
MEM - fetch/store memory operand (maybe)
WB - write to register (maybe)

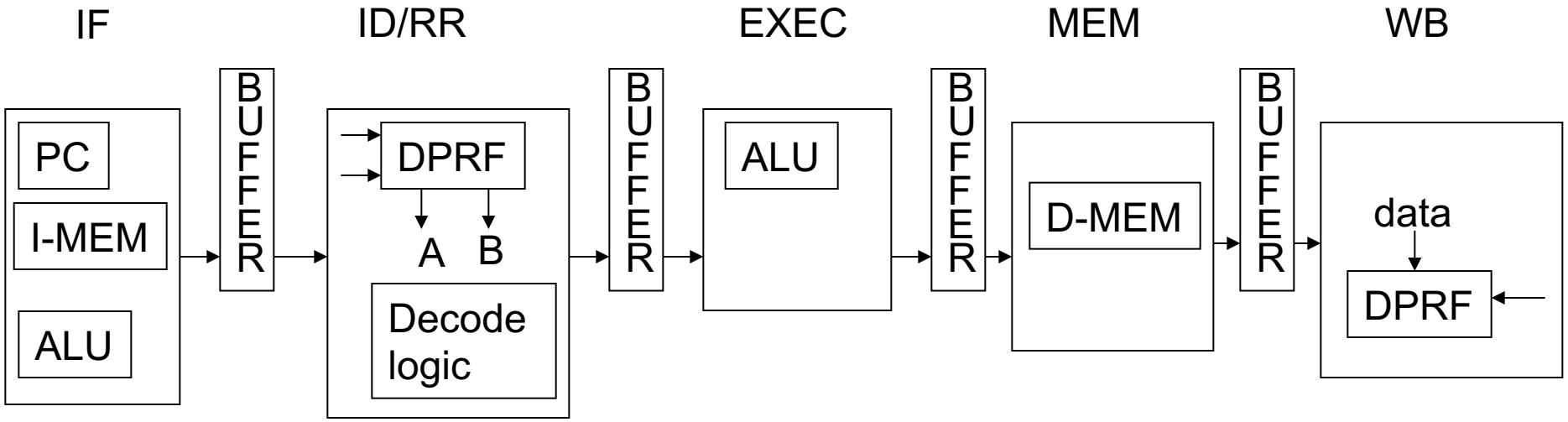
Again: What does each stage need?



STAGE	Units in Use
FETCH	ALU, PC, MEM
ID/RR	Reg-file, decode logic
EXEC	ALU
MEM	MEM
WB	Reg-file

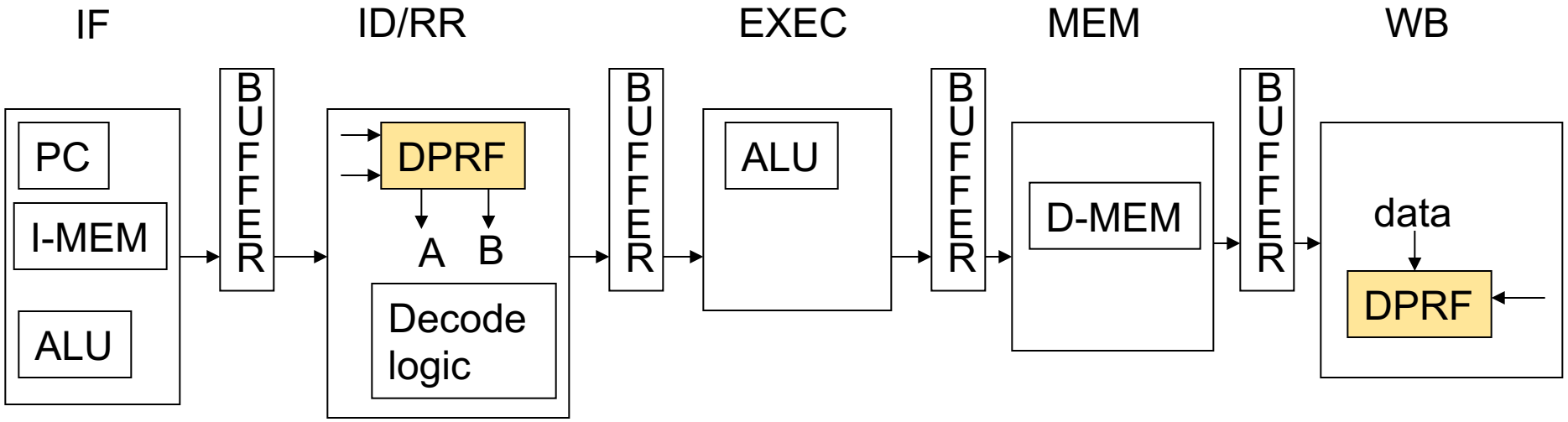
- Conclusion: we need a little datapath in each stage

A shiny new data path!



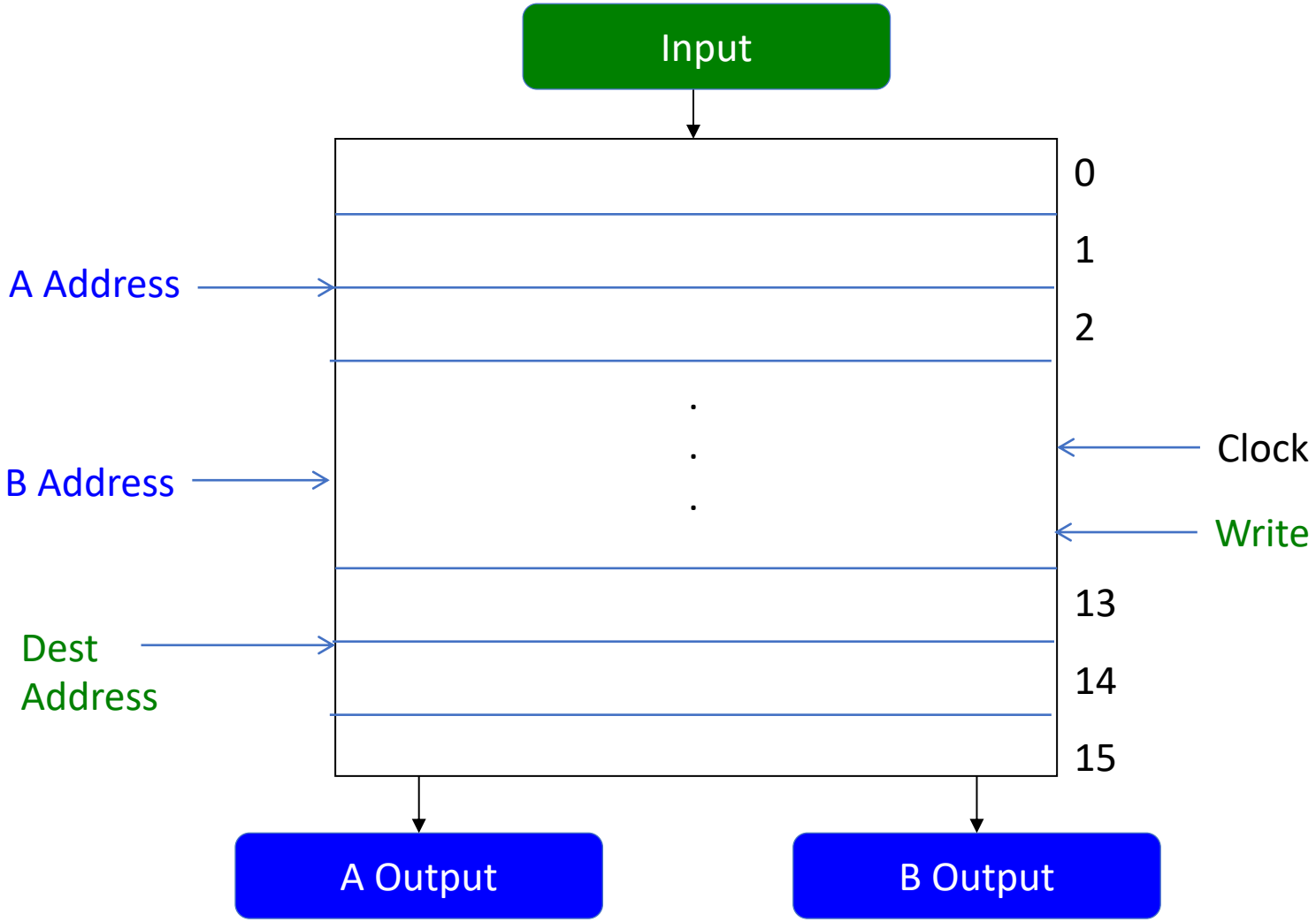
STAGE	**Units in Use**
FETCH	**ALU, PC, MEM**
ID/RR	**Reg-file, decode logic**
EXEC	**ALU**
MEM	**MEM**
WB	**Reg-file**

Two register files? Not.

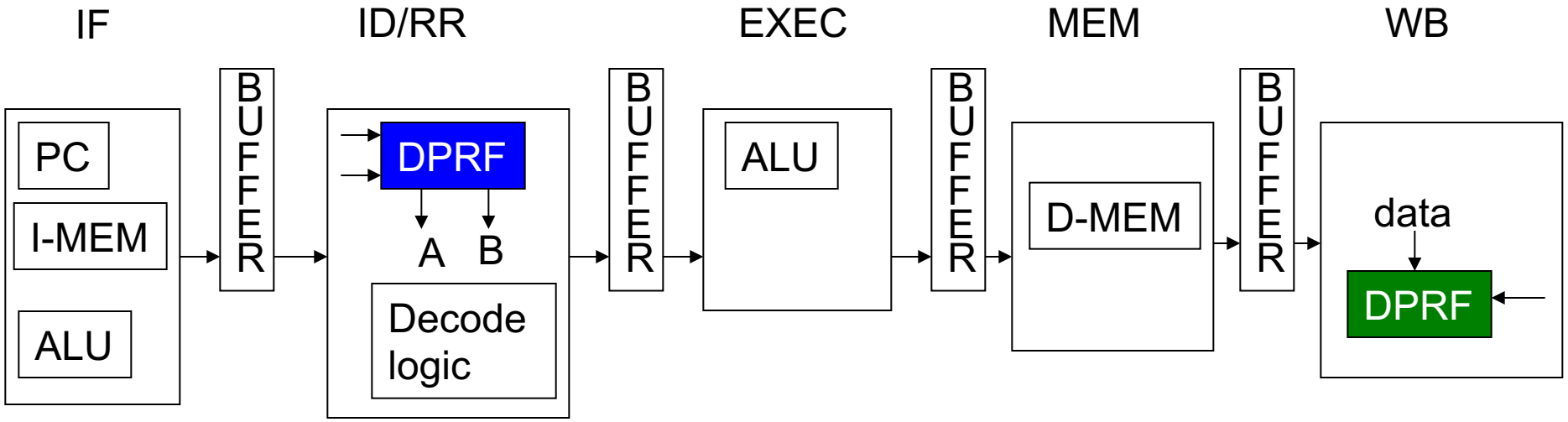


- We use a register file in two stages

Do you remember this device?

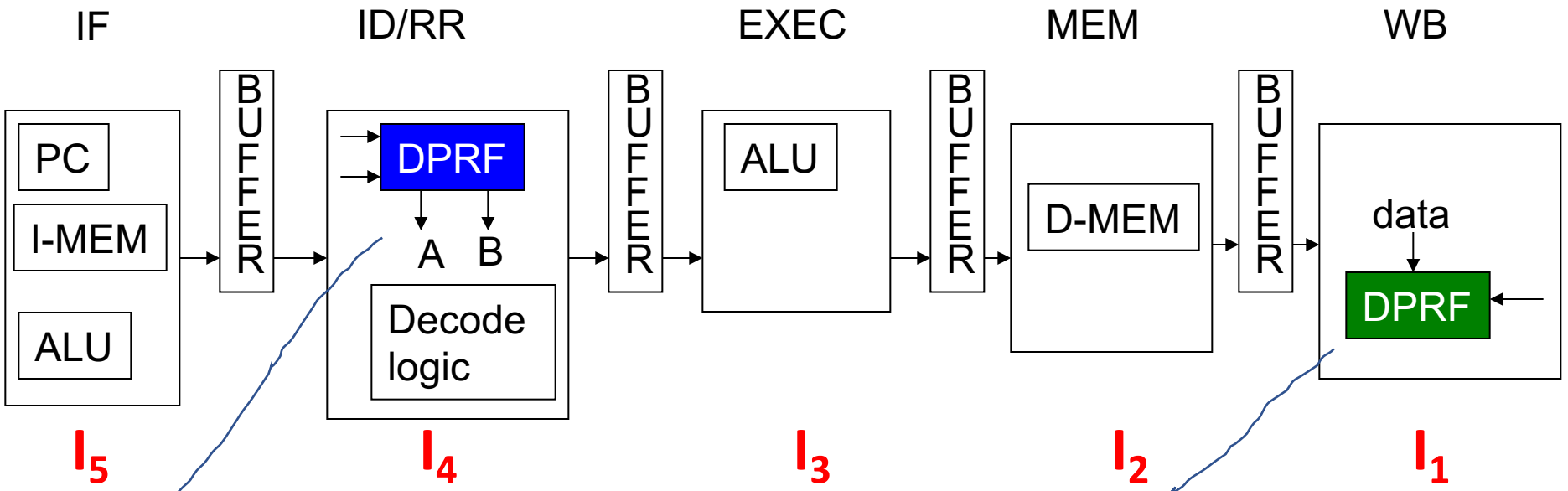


Two register files? Not.



- We use a register file in two stages, but conveniently
  - ID/RR uses read ports (A addr, B addr B, A output, B output)
  - WB uses write port (Dest addr, Input)
- So we can happily share the same register file on each clock cycle

Example of sharing the register file

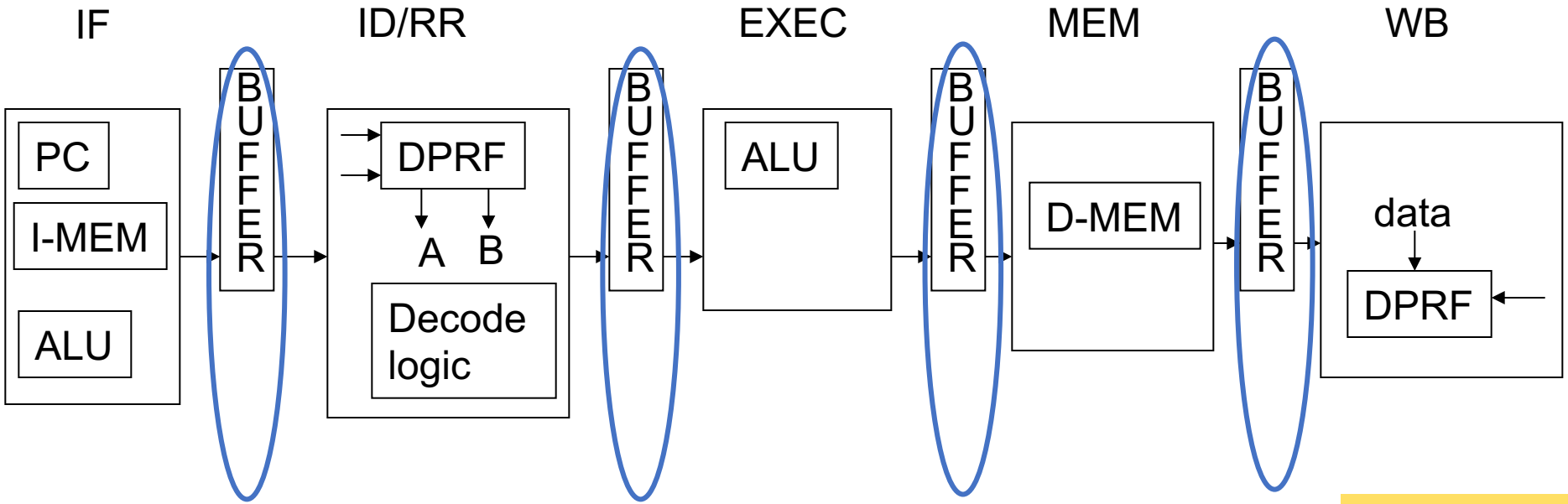


- I_1: ADD R_1, R_2, R_3
- I_4: ADD R_5, R_6, R_7

$R_1 \leftarrow R_2 + R_3$

$R_5 \leftarrow R_6 + R_7$

But what about those buffers?



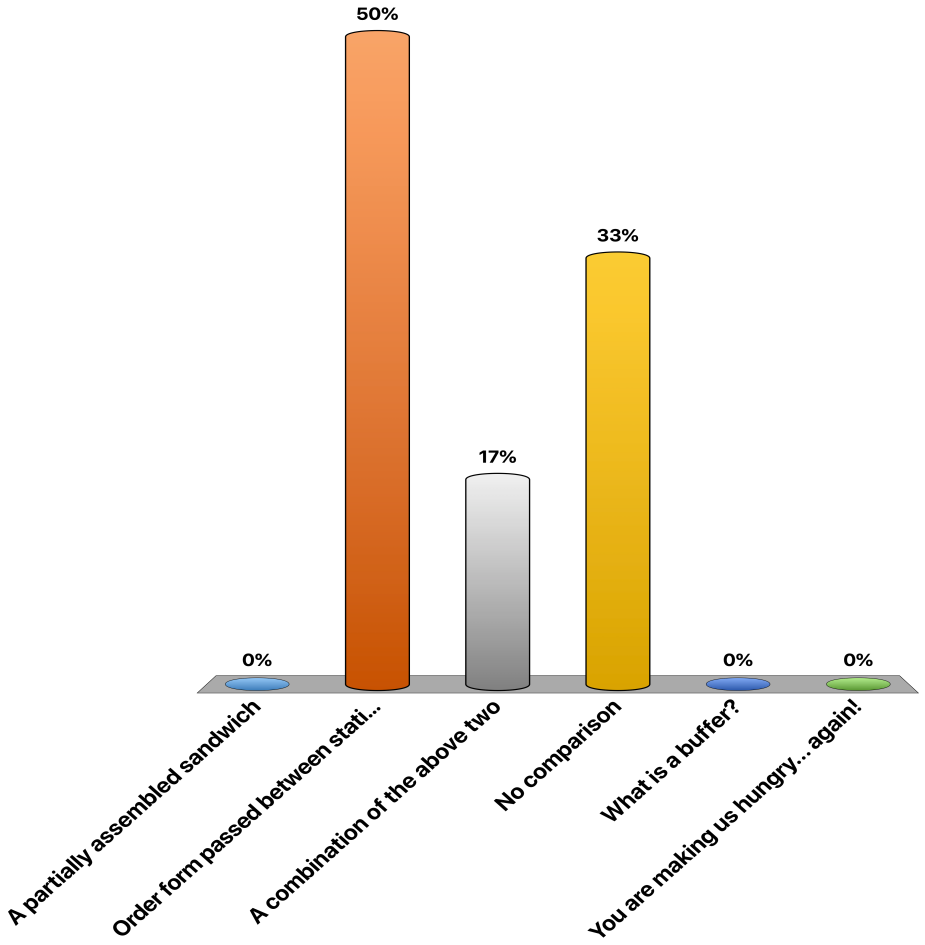
- What goes in those buffers? Same? Different?
- The different stages seem to need different information
- What was that sandwich analogy?

Session ID: dag2200

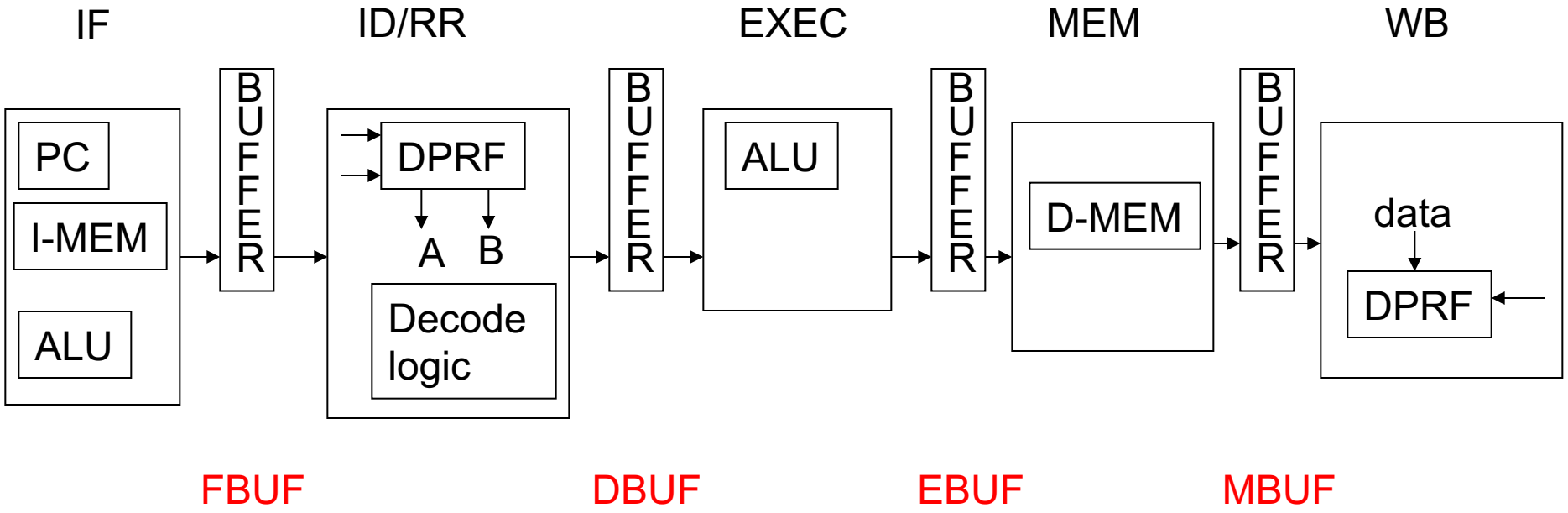
Comparing the instruction pipeline…

…to Bill's sandwich pipeline, the buffers between stages serve the same function as

A. A partially assembled sandwich
B. Order form passed between stations
C. A combination of the above two
D. No comparison
E. Buffer? You mean buffet?
F. You are making us hungry... again!

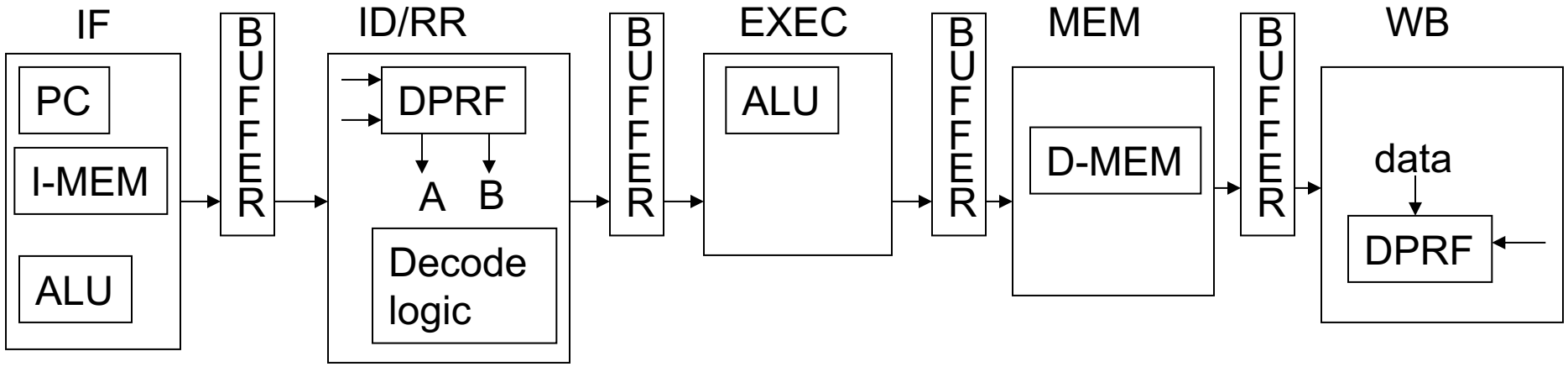


So what makes up our partially assembled data sandwich?



- The IR (or parts of it) is the order form
- The data sandwich is the partial execution results
- Do the buffers have to all carry the same information? Or can they be different?

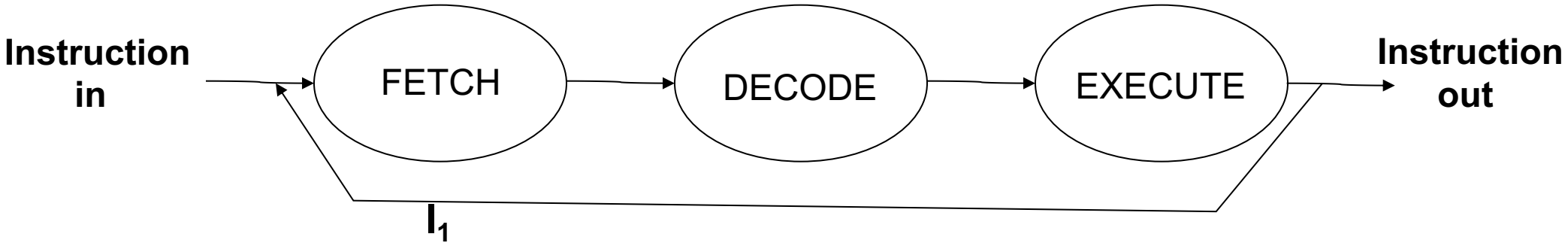
What does each stage need in the buffers?



FBUF DBUF EBUF MBUF

Name	Output of Stage	Contents
FBUF	IF	Primarily contains instruction read from memory
DBUF	ID/RR	Decoded IR and values read from register file
EBUF	EX	Primarily contains result of ALU operation plus other parts of the instruction depending on the instruction specifics
MBUF	MEM	Same as EBUF if instruction is not LW or SW; If instruction is LW, then buffer contains the contents of memory location read

Our non-pipelined LC-2200



At each point, the processor is in exactly one macro state

What "macro state" is a pipelined processor in at any point in time?

A. Fetch
B. Decode
C. Execute
D. All of the above
E. We need a new "sandwich" state

Anatomy of an ADD instruction

- IF stage (cycle 1)
- ID/RR stage (cycle 2)
- EX stage (cycle 3)
- MEM stage (cycle 4)
- WB stage (cycle 5)

Actions for an ADD instruction

Stage	Actions	Comments
IF	MEM[PC] → FBUF.IR PC + 1 → PC	Instruction is fetched from memory and placed in FBUF (which is essentially the IR)
ID/RR	DPRF[FBUF.IR.Ry] → DBUF.A DPRF[FBUF.IR.Rz] → DBUF.B FBUF.IR.Opcode → DBUF.Opcode FBUF.IR.Rx → DBUF.Rx	Read register Ry into DBUF.A Read register Rz into DBUF.B Copy opcode from FBUF to DBUF Copy Rx from FBUF to DBUF
EX	DBUF.A + DBUF.B → EBUF.Result DBUF.OPCODE → EBUF.Opcode DBUF.Rx → EBUF.Rx	Perform addition Copy opcode from DBUF to EBUF Copy Rx from DBUF to EBUF
MEM	EBUF → MBUF	Copy EBUF to MBUF
WB	MBUF.Result → DPRF[MBUF.Rx]	Write the result of addition into register Rx

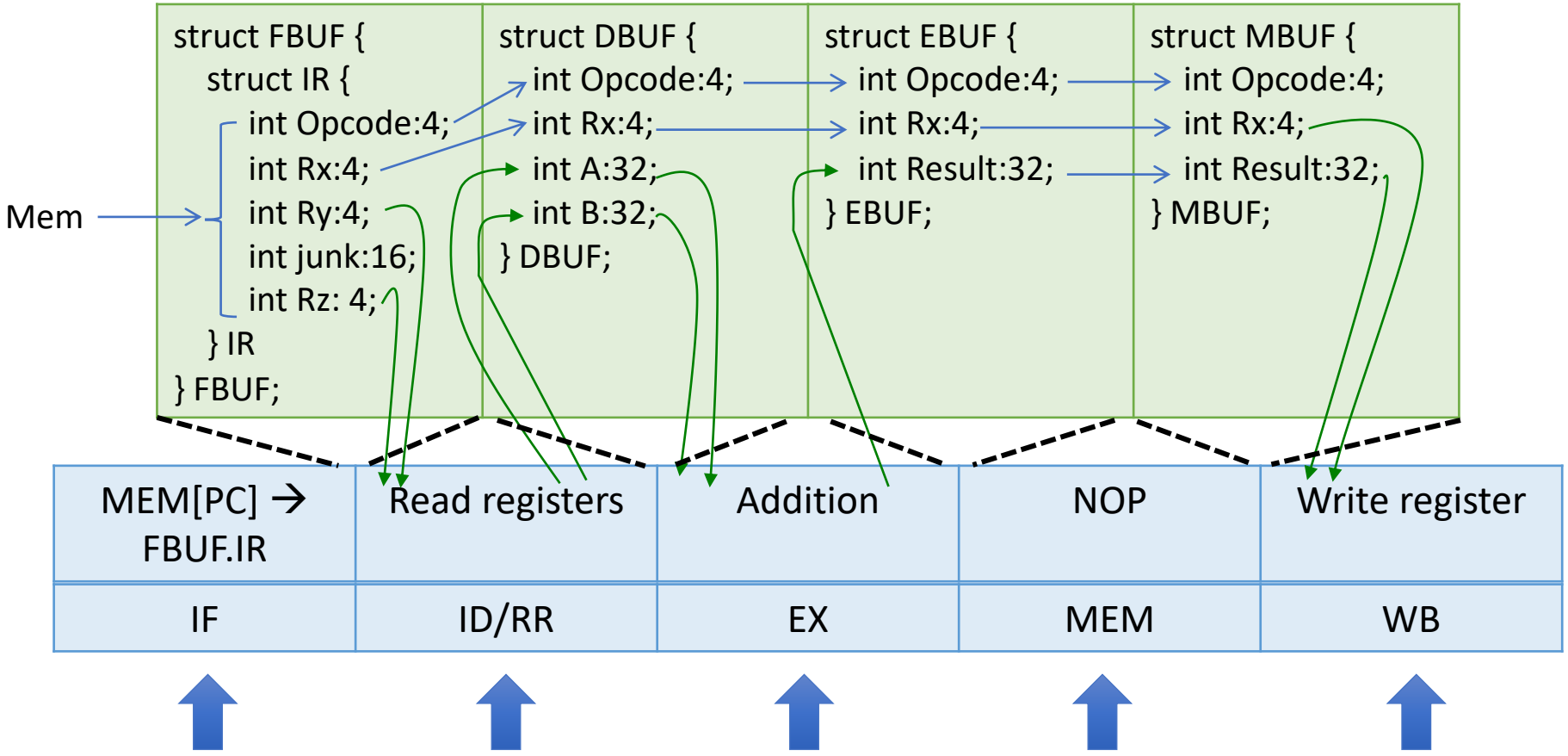
Actions for an ADD instruction

Stage	Actions	Comments
IF	MEM[PC] → FBUF.IR PC + 1 → PC	Instruction is fetched from memory and placed
ID/RR	DPRF[FBUF.IR.Ry] → DBUF.A DPRF[FBUF.IR.Rz] → DBUF.B FBUF.IR.Opcode → DBUF.Opcode FBUF.IR.Rx → DBUF.Rx	
EX	DBUF.A + DBUF.B → EBUF.Result DBUF.OPCODE → EBUF.Opcode DBUF.Rx → EBUF.Rx	dition de from DBUF to EBUF m DBUF to EBUF
MEM	EBUF → MBUF	to MBUF
WB	MBUF.Result → DPRF[MBUF.Rx]	Write the result of addition into register Rx

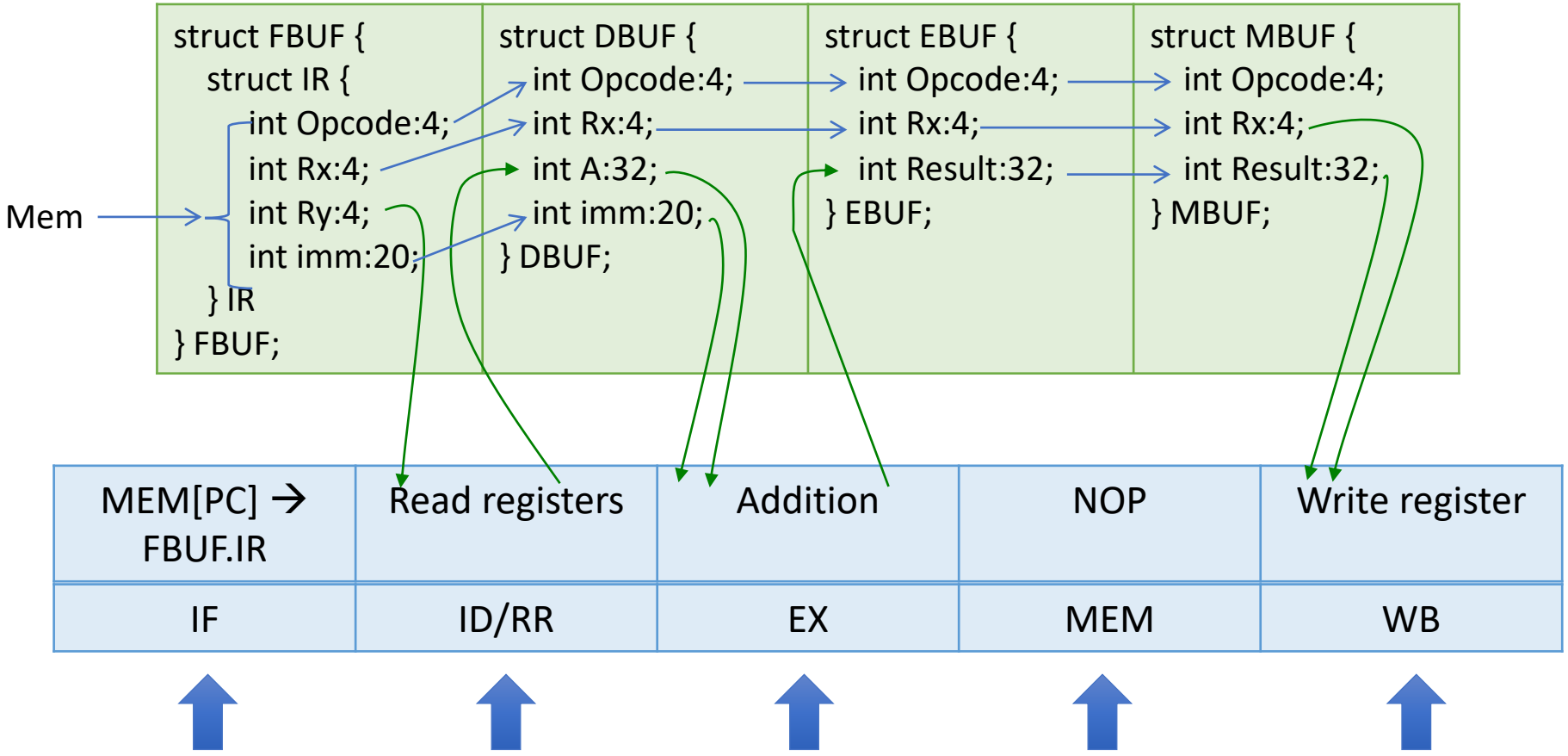
Considering only the Add instruction, quantify the sizes of the various buffers between the stages of the pipeline.

FBUF?
DBUF?
EBUF?
MBUF?

How does this pipeline work for ADD?



How does this pipeline work for ADDI?



How to generalize design of pipeline register?

- Look at each instruction in the ISA
- Decide what needs to be carried over from one stage to the next

Design the DBUF register for LC-2200

- Don't optimize by overloading the different fields of the register
- Decide what fields need to be in DBUF for each instruction in the ISA

ADD/NAND	ADDI	BEQ	And so on…
struct DBUF { int Opcode:4; int Rx:4; int A:32; int B:32; } DBUF;	struct DBUF { int Opcode:4; int Rx:4; int A:32; int imm:20; } DBUF;	struct DBUF { int Opcode:4; int Rx:4; int imm:20; int PC:32; } DBUF;	

Now take the UNION of the requirements

ADD/NAND	ADDI	BEQ	And so on…
struct DBUF { int Opcode:4; int Rx:4; int A:32; int B:32; } DBUF;	struct DBUF { int Opcode:4; int Rx:4; int A:32; int imm:20; } DBUF;	struct DBUF { int Opcode:4; int Rx:4; int imm:20; int PC:32; } DBUF;	

```
struct DBUF {
    int Opcode:4;
    int Rx:4;
    int A:32;
    int B:32;
    int imm:20;
    int PC:32;
} DBUF;
```

Now take the UNION of the requirements

```
struct DBUF {
   int Opcode:4;
   int Rx:4;
   int A:32;
   int B:32;
   int imm:20;
   int PC:32;
} DBUF;
```

- In the ID/RR stage, fill in fields depending on the instruction in FBUF.IR
- Leave unneeded fields unfilled
- What fields would we fill in for

  JALR $at, $ra # $ra ← PC; PC ← $at

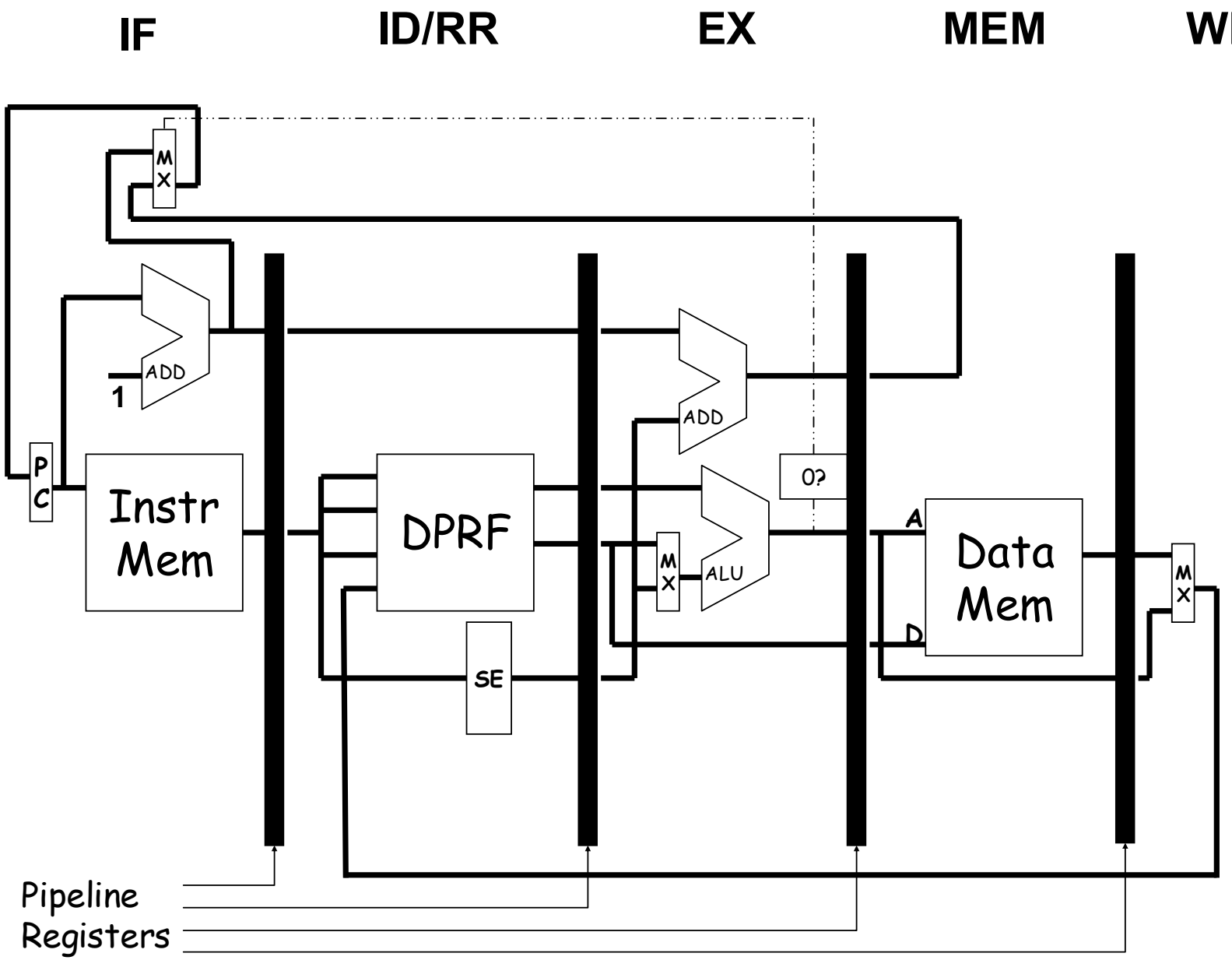
Session ID: dag2200

For the BEQ instruction, the fields in DBUF *not filled in…*

BEQ Rx, Ry, offset ; If (Rx == Ry) PC ←PC + offset

A. A (contents of Ry)
B. B (contents of Rx)
C. PC
D. Immediate value
E. Rx specifier

```
struct DBUF {
   int Opcode:4;
   int Rx:4;
   int A:32;
   int B:32;
   int imm:20;
   int PC:32;
} DBUF;
```


IF
ID/RR
EX
MEM
WB
M
X
ADD
1
P
C
Instr
Mem
DPRF
SE
ADD
M
X
ALU
0?
A
D
Data
Mem
M
X
Pipeline
Registers