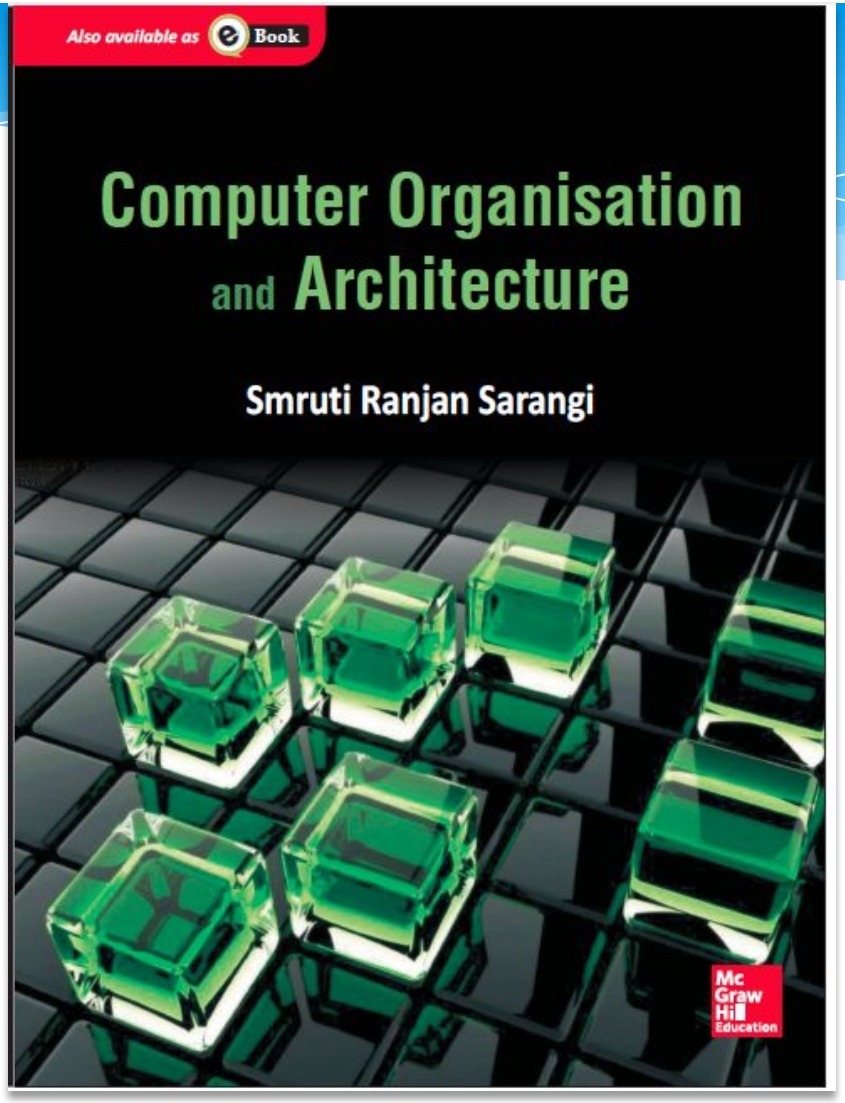
PowerPoint Slides

Computer Organisation and Architecture

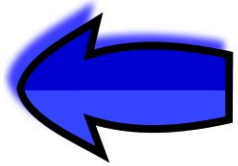
Smruti Ranjan Sarangi,
IIT Delhi

Chapter 8 Processor Design



These slides are meant to be used along with the book: Computer Organisation and Architecture, Smruti Ranjan Sarangi, McGrawHill 2015
Visit: http://www.cse.iitd.ernet.in/~srsarangi/archbooksoft.html

Outline

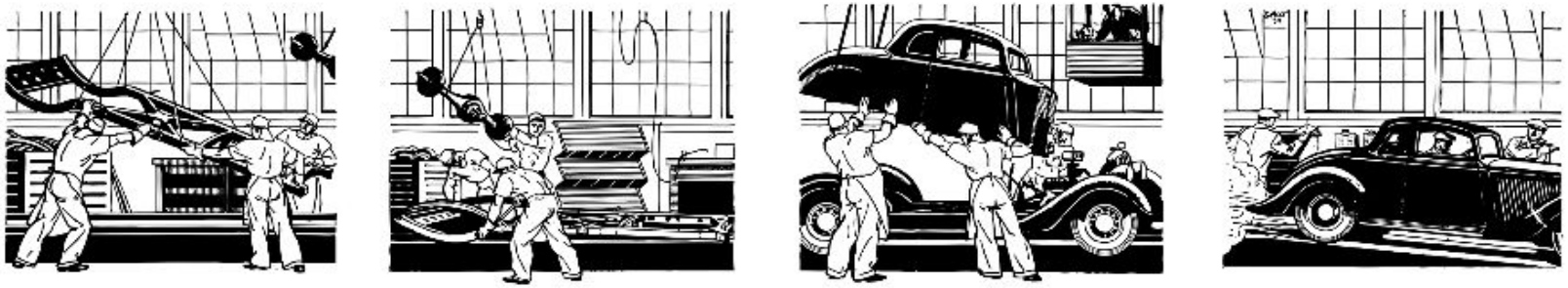


* Overview of a Processor
* Detailed Design of each Stage
* The Control Unit
* Microprogrammed Processor
* Microassembly Language
* The Microcontrol Unit

Processor Design

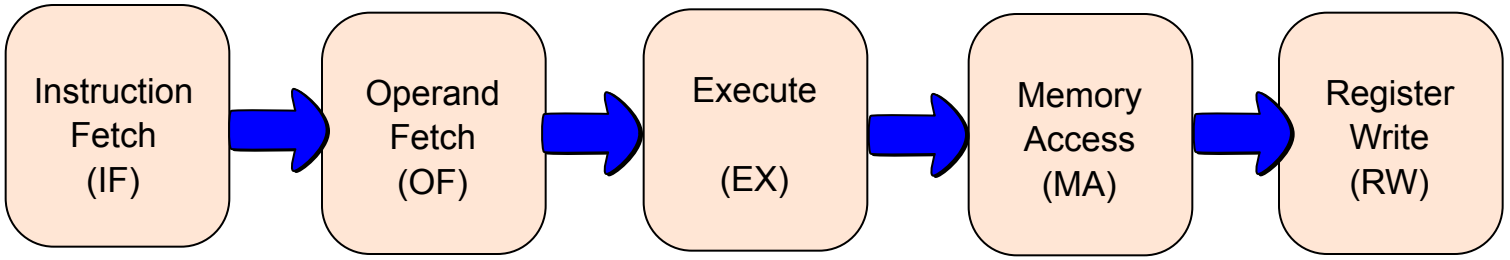
* The aim of processor design
  * Implement the entire SimpleRisc ISA
  * Process the binary format of instructions
  * Provide as much of performance as possible
* Basic Approach
  * Divide the processing into stages
  * Design each stage separately

A Car Assembly Line



* Similar to a car assembly line
  * Cast raw metal into the chassis of a car
  * Build the Engine
  * Assemble the engine and the chassis
  * Place the dashboard, and upholstery

A Processor Divided Into Stages



* Instruction Fetch (IF)
  * Fetch an instruction from the instruction memory
  * Compute the address of the next instruction

Operand Fetch (OF) Stage

* Operand Fetch (OF)
  * Decode the instruction (break it into fields)
  * Fetch the register operands from the register file
  * Compute the branch target (PC + offset)
  * Compute the immediate (16 bits + 2 modifiers)
  * Generate control signals (we will see later)

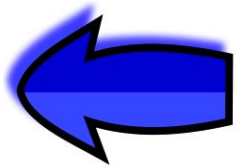
Execute (EX) Stage

* The EX Stage
  * Contains an Arithmetic-Logical Unit (ALU)
    * This unit can perform all arithmetic operations (add, sub, mul, div, cmp, mod), and logical operations (and, or, not)
  * Contains the branch unit for computing the branch condition (beq, bgt)
  * Contains the flags register (updated by the cmp instruction)

MA and RW Stages

* MA (Memory Access) Stage
  * Interfaces with the memory system
  * Executes a load or a store
* RW (Register Write) Stage
  * Writes to the register file
  * In the case of a call instruction, it writes the return address to register, ra

Outline

* Outline of a Processor
* Detailed Design of each Stage 
* The Control Unit
* Microprogrammed Processor
* Microassembly Language
* The Microcontrol Unit

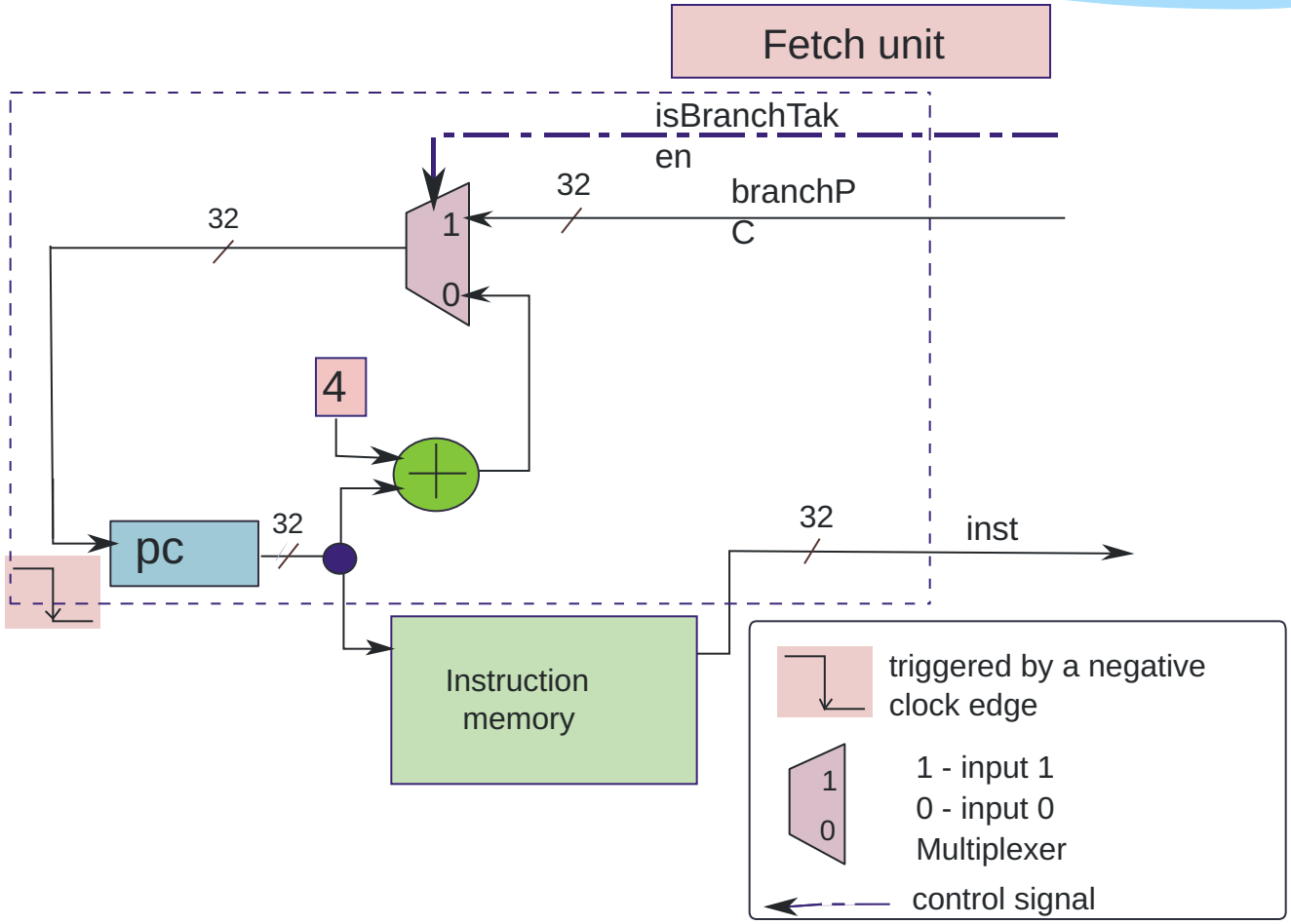
Design Goals

Functional Completeness and Correctness

Performance | **Power** | **Area**

Simplicity
Ease of Analysis and Debugging

Instruction Fetch (IF) Stage



The Fetch unit

* The pc register contains the program counter (negative edge triggered)
* We use the pc to access the instruction memory
* The multiplexer chooses between
  * pc + 4
  * branchTarget
* It uses a control signal → isBranchTaken

isBranchTaken

* isBranchTaken is a control signal
  * It is generated by the EX unit
* Conditions on isBranchTaken

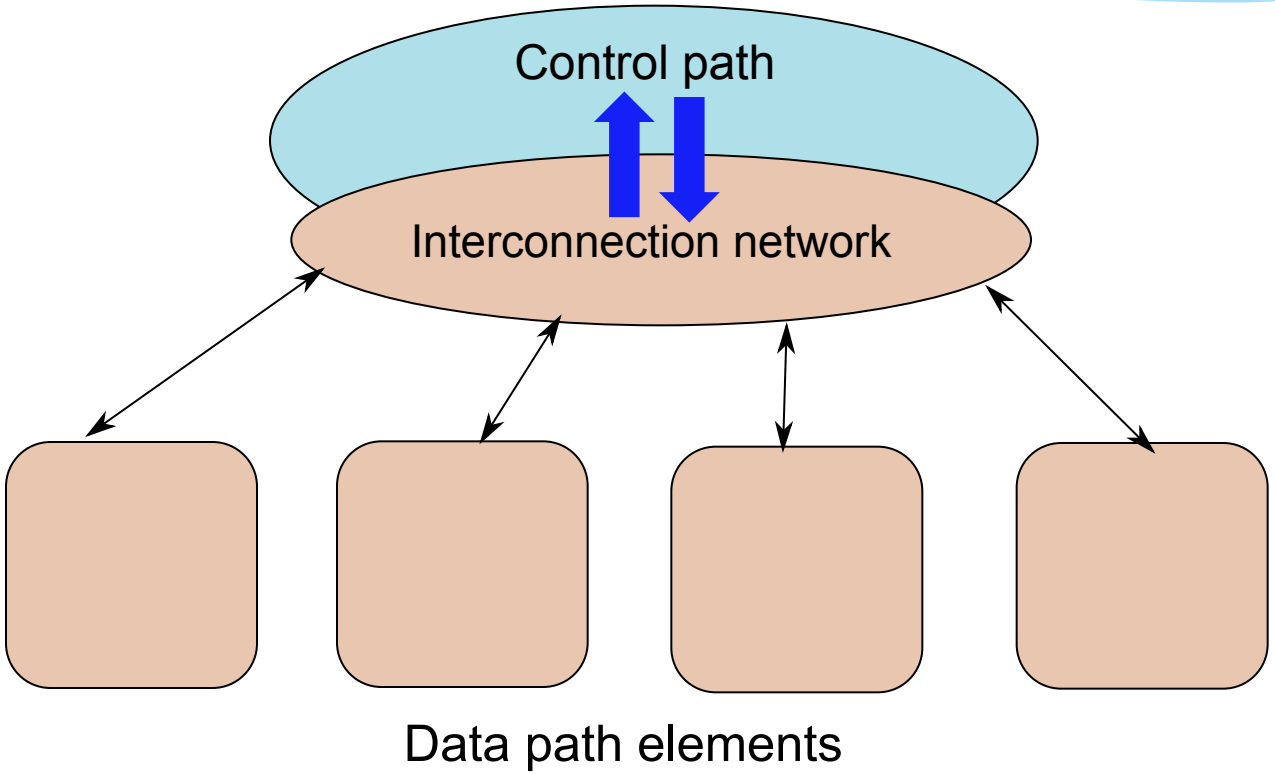
Instruction	Value of *isBranchTaken*
non-branch instruction	0
call	1
ret	1
b	1
beq	branch taken – 1 branch not taken – 0
bgt	branch taken – 1 branch not taken – 0

Data Path and Control Path

* The data path consists of all the elements in a processor that are dedicated to storing, retrieving, and processing data such as register files, memory, and the ALU.

* The control path primarily contains the control unit, whose role is to generate the appropriate signals to control the movement of instructions, and data in the data path.

Control Path



We will currently look at the hardwired control path.

Operand Fetch Unit

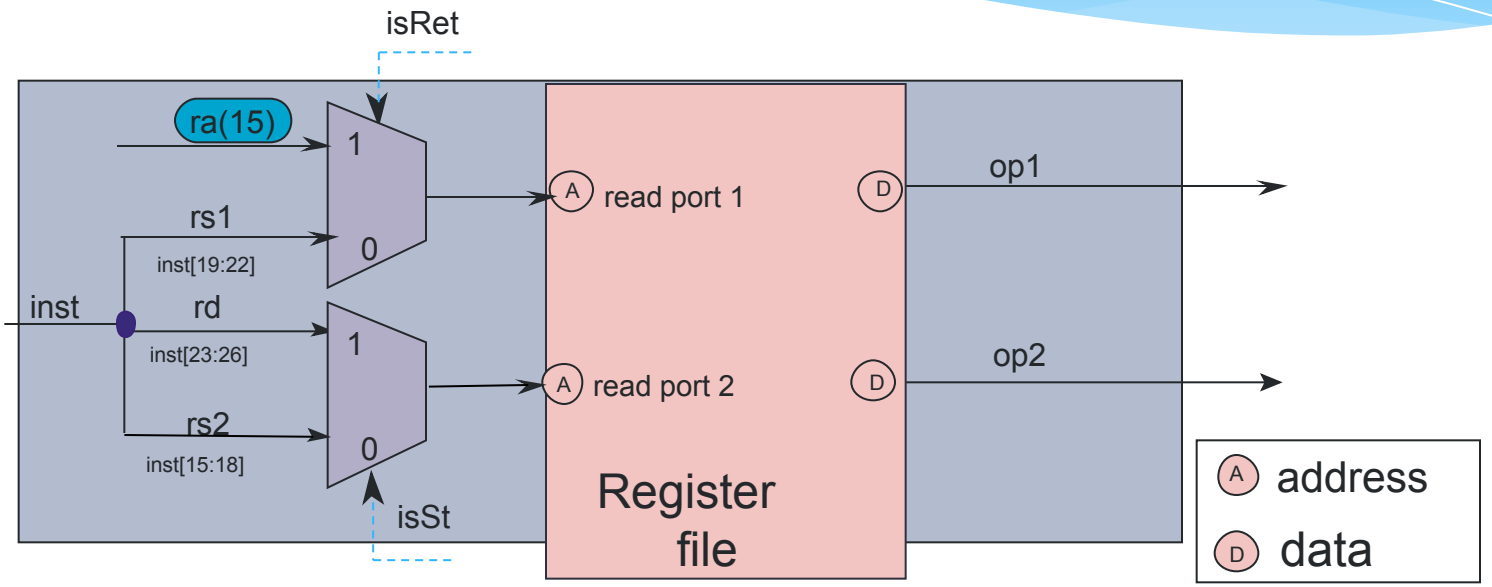
Inst.	Code	Format	Inst.	Code	Format
add	00000	add rd, rs1, (rs2/imm)	lsl	01010	lsl rd, rs1, (rs2/imm)
sub	00001	sub rd, rs1, (rs2/imm)	lsr	01011	lsr rd, rs1, (rs2/imm)
mul	00010	mul rd, rs1, (rs2/imm)	asr	01100	asr rd, rs1, (rs2/imm)
div	00011	div rd, rs1, (rs2/imm)	nop	01101	nop
mod	00100	mod rd, rs1, (rs2/imm)	ld	01110	ld rd, imm[rs1]
cmp	00101	cmprs1, (rs2/imm)	st	01111	st rd, imm[rs1]
and	00110	and rd, rs1, (rs2/imm)	beq	10000	beq offset
or	00111	or rd, rs1, (rs2/imm)	bgt	10001	bgt offset
not	01000	not rd, (rs2/imm)	b	10010	b offset
mov	01001	mov rd, (rs2/imm)	call	10011	call offset
			ret	10100	ret

Instruction Formats

Format	Definition				
branch	*op* (28-32)	*offset* (1-27)			
register	*op* (28-32)	*I* (27)	*rd* (23-26)	*rs1* (19-22)	*rs2* (15-18)
immediate	*op* (28-32)	*I* (27)	*rd* (23-26)	*rs1* (19-22)	*imm* (1-18)
op → opcode, *offset* → branch offset, *I* → immediate bit, *rd* → destination register					
*rs*1 → source register 1, *rs*2 → source register 2, *imm* → immediate operand					

* Each format needs to be handled separately.

Register File Read

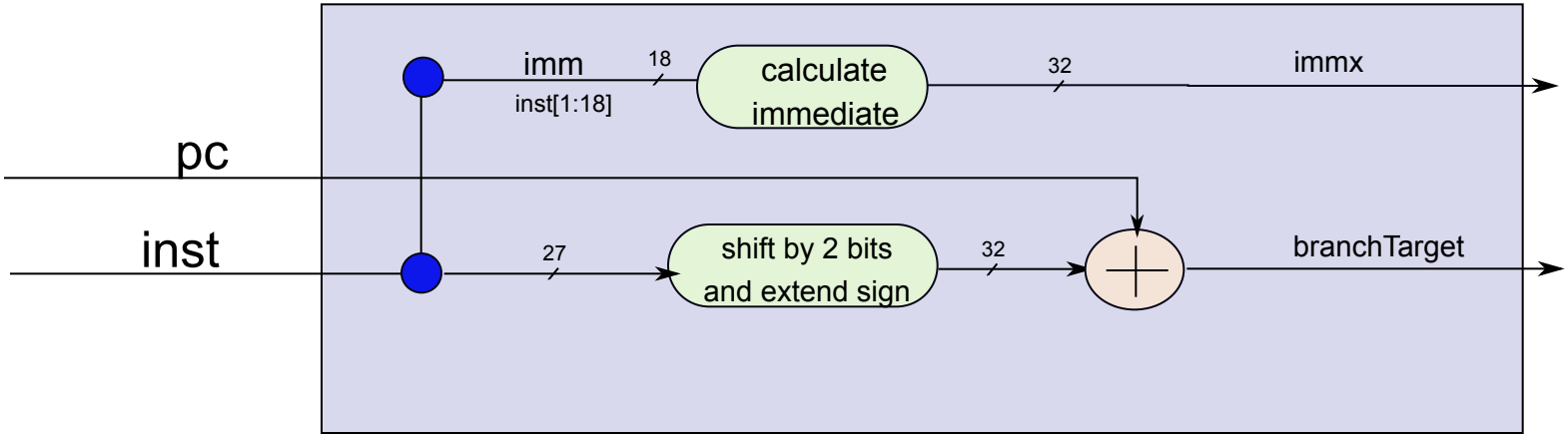


* First input → rs1 or ra(15) (ret instruction)
* Second input → rs2 or rd (store instruction)

Register File Access

* The register file has two read ports
  * 1st Input
  * 2nd Input
* The two outputs are op1, and op2
  * op1 is the branch target (return address) in the case of a ret instruction, or rs1
  * op2 is the value that needs to be stored in the case of a store instruction, or rs2

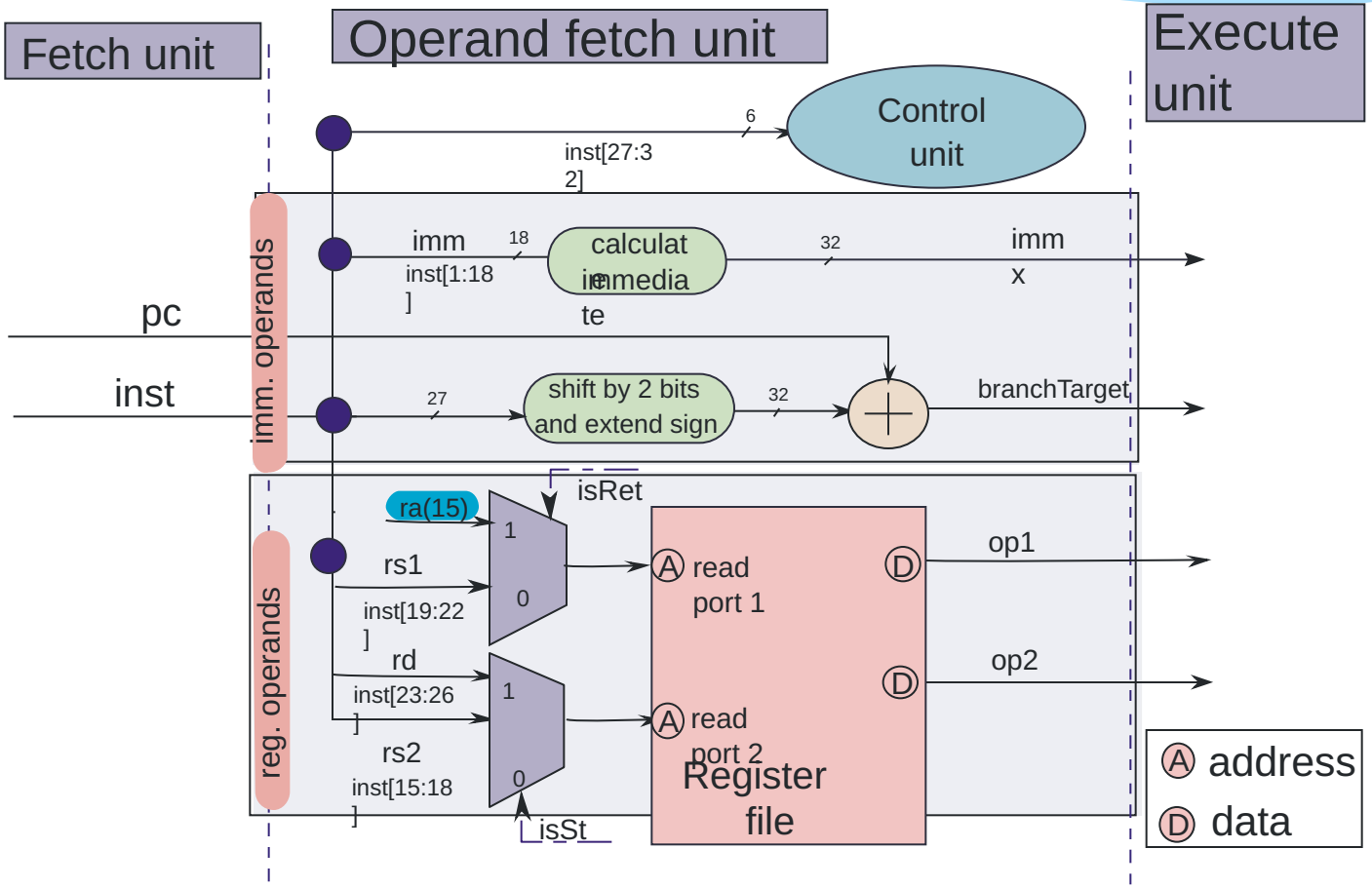
Immediate and Branch Unit



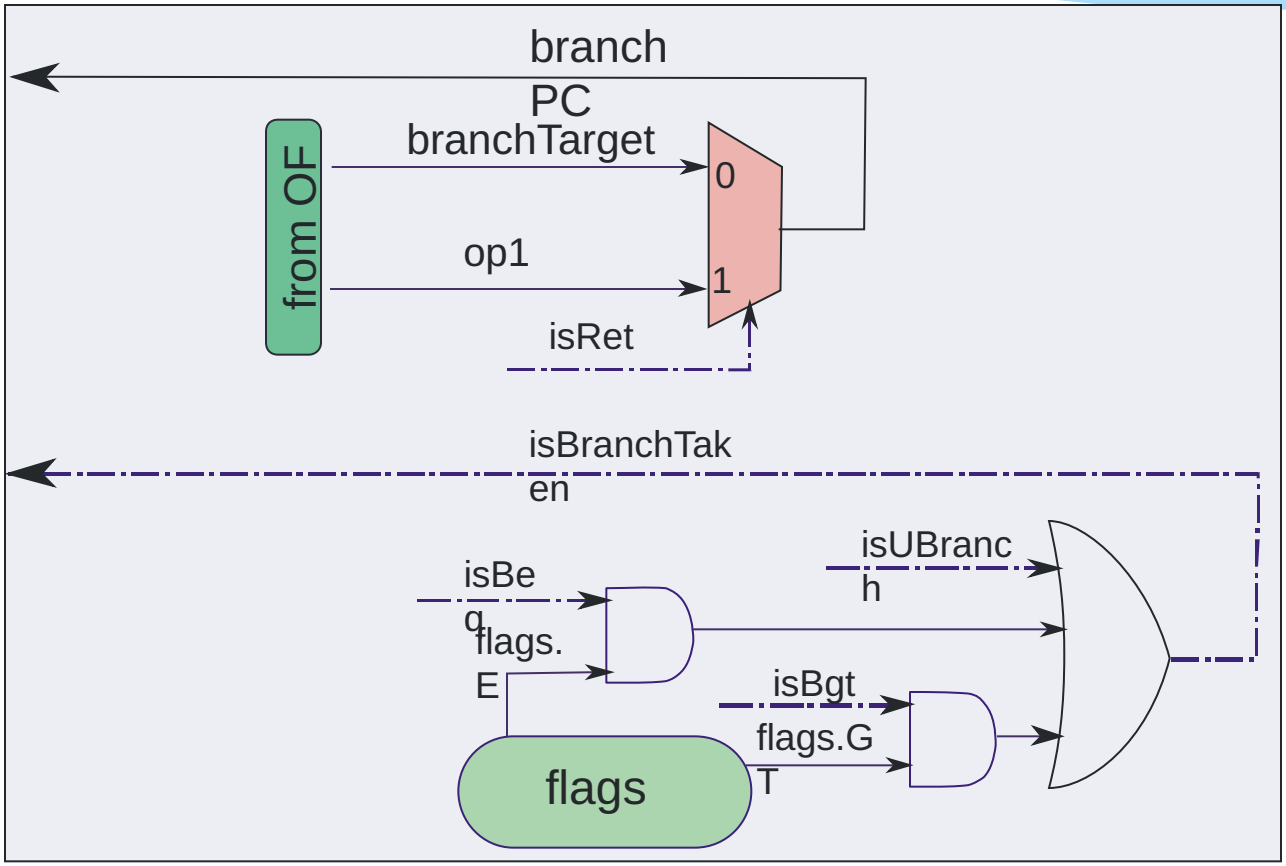
* Compute immx (extended immediate), branchTarget, irrespective of the instruction format.
* For the branchTarget we need to choose between the embedded target and op1 (ret)

(opcode, I bit)

OF Unit

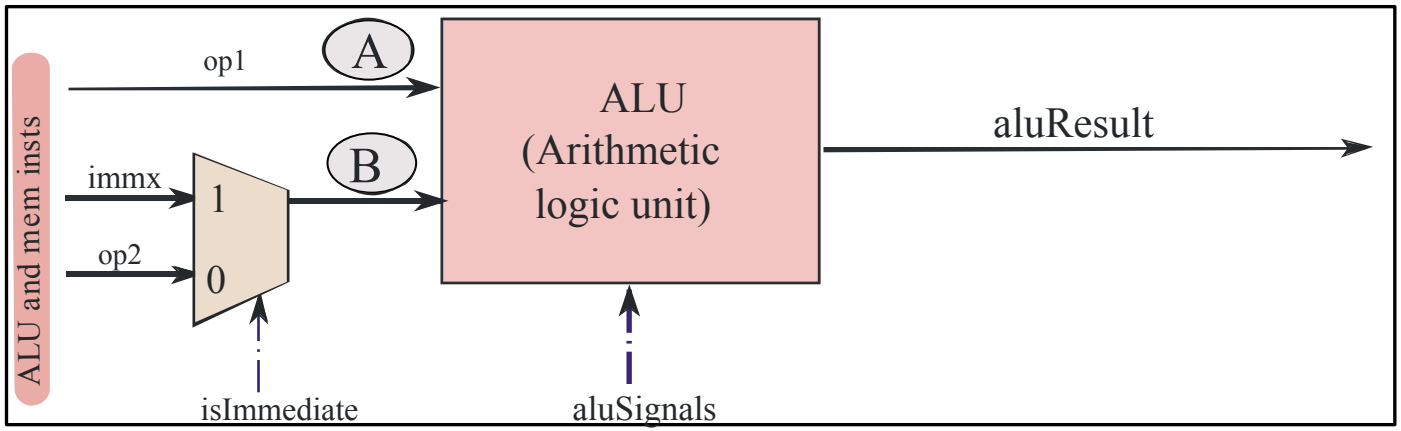


EX Stage – Branch Unit



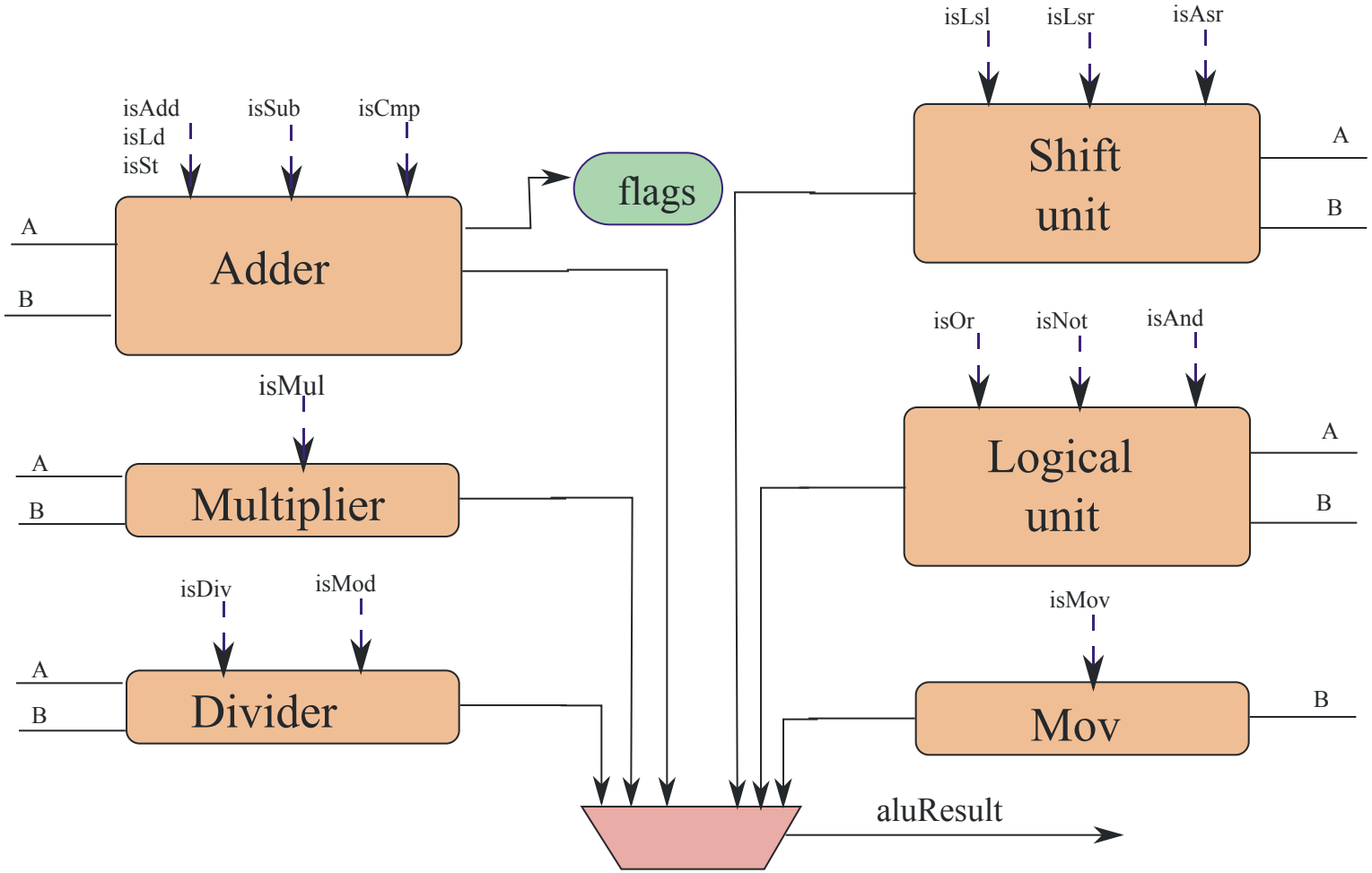
Generates the isBranchTaken Signal

ALU



Choose between immx and op2 based on the value of the I bit

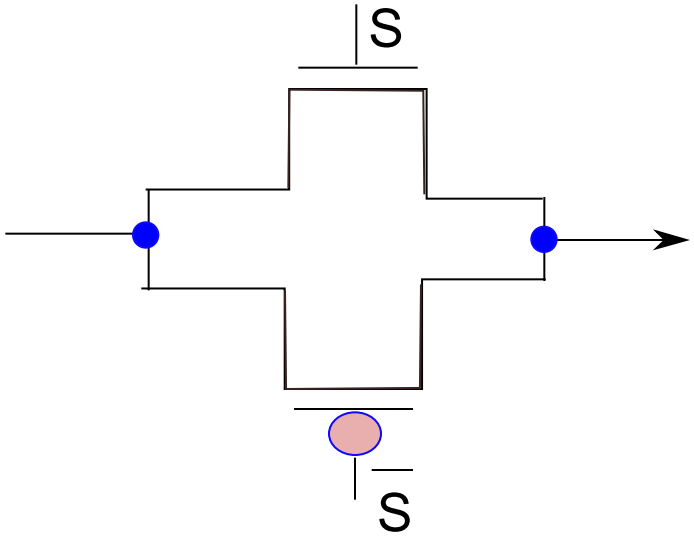
Inside the ALU



Disabling some Inputs

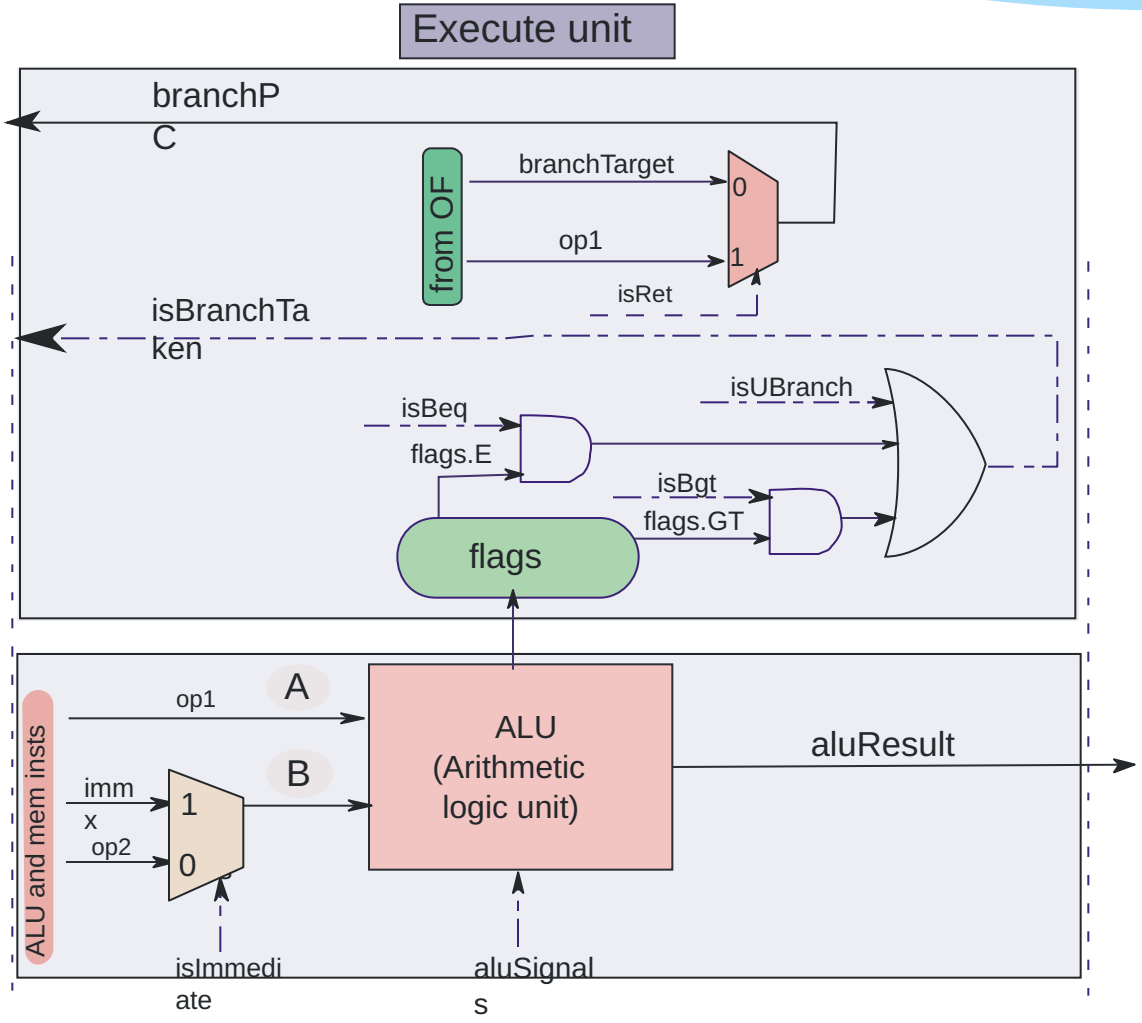
* We do not want all the units of the ALU to be active at the same time because of we want to save power
* The instruction will only use 1 unit
* Power is dissipated when the inputs or outputs make a transition ($0 \rightarrow 1$, $1 \rightarrow 0$)
  * We shall avoid a transition by not letting the new inputs to propagate to units that do not require them
  * They will thus have the old inputs (no switching)

Use a Transmission Gate

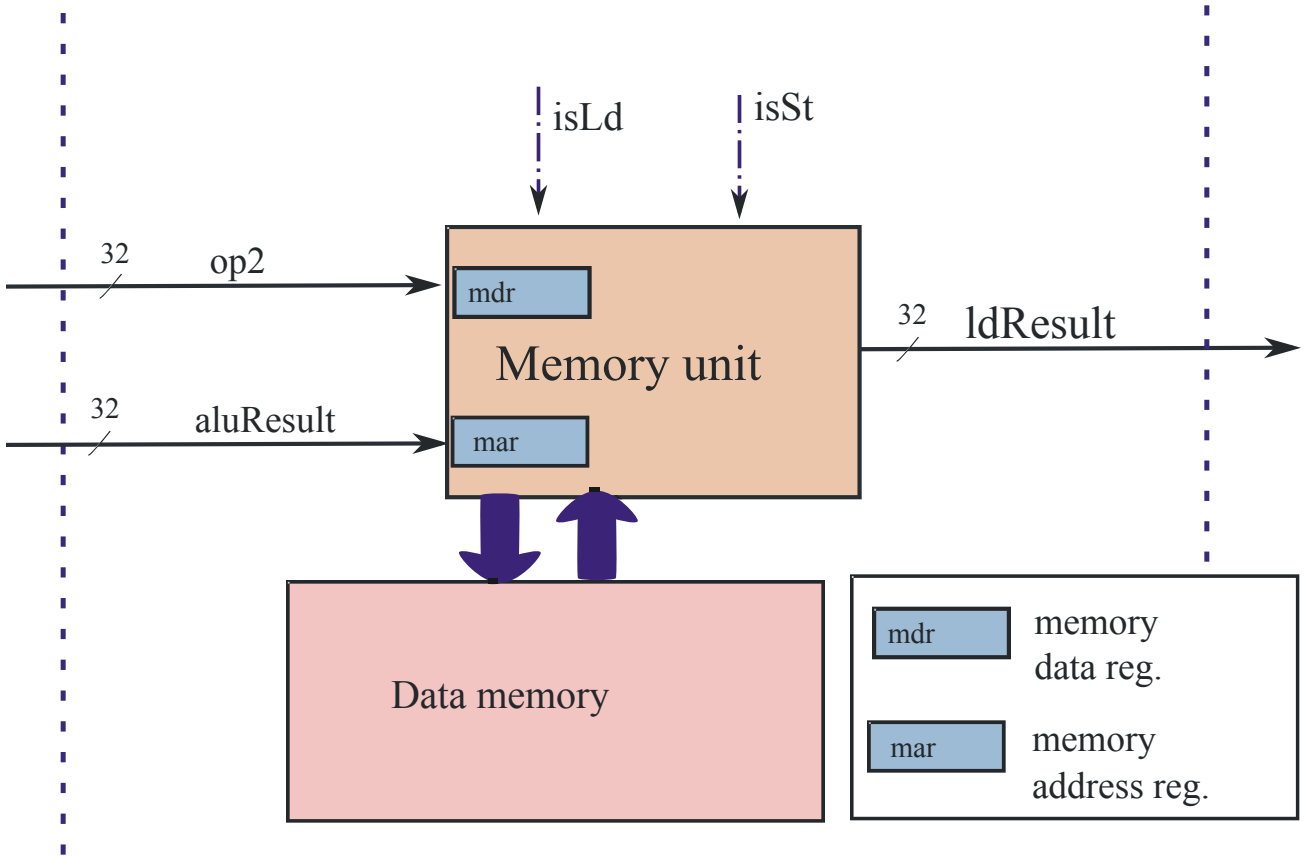


* output = input (if S = 1)
* Otherwise, the output is totally disconnected from the input

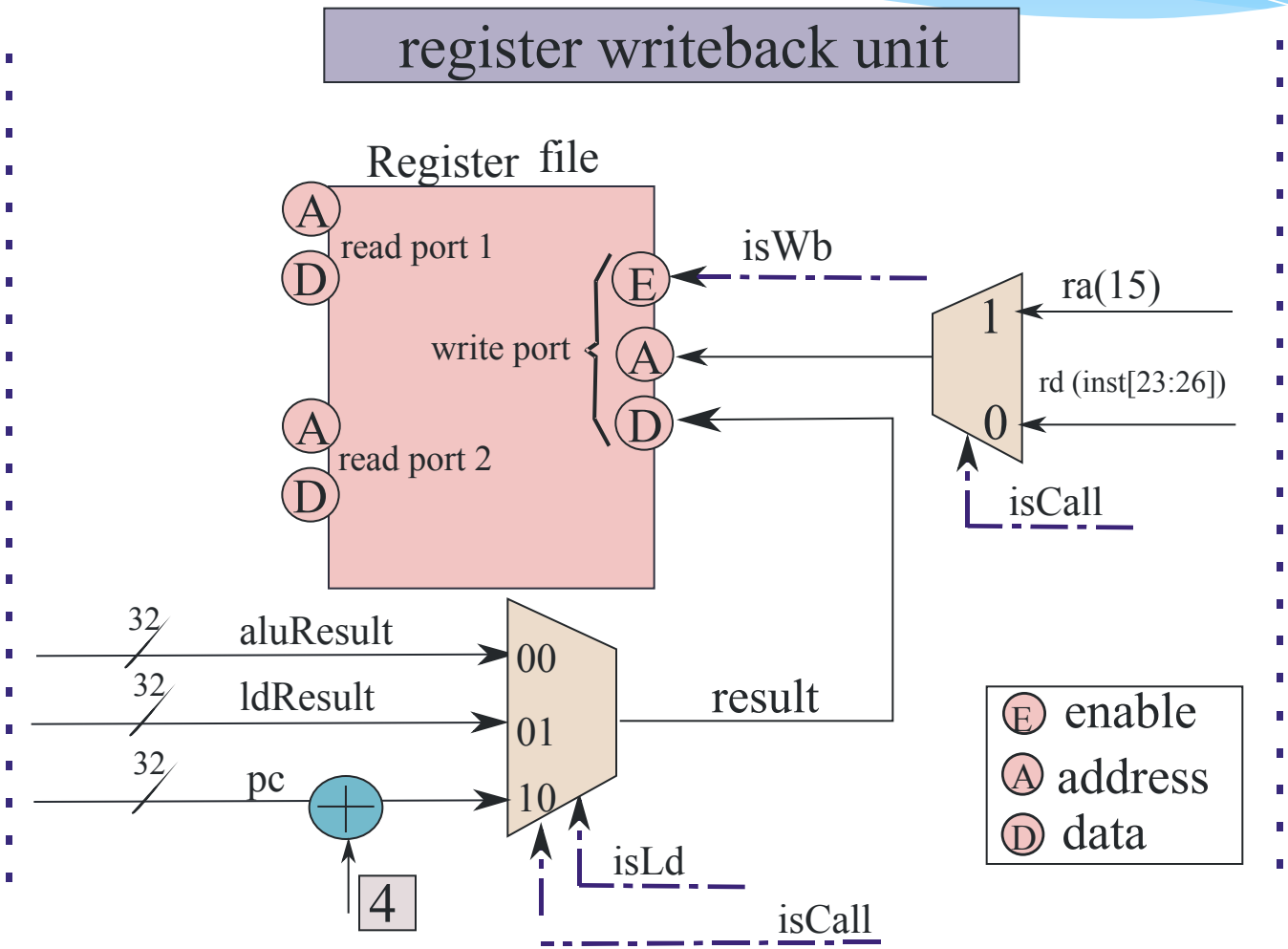
EX Unit

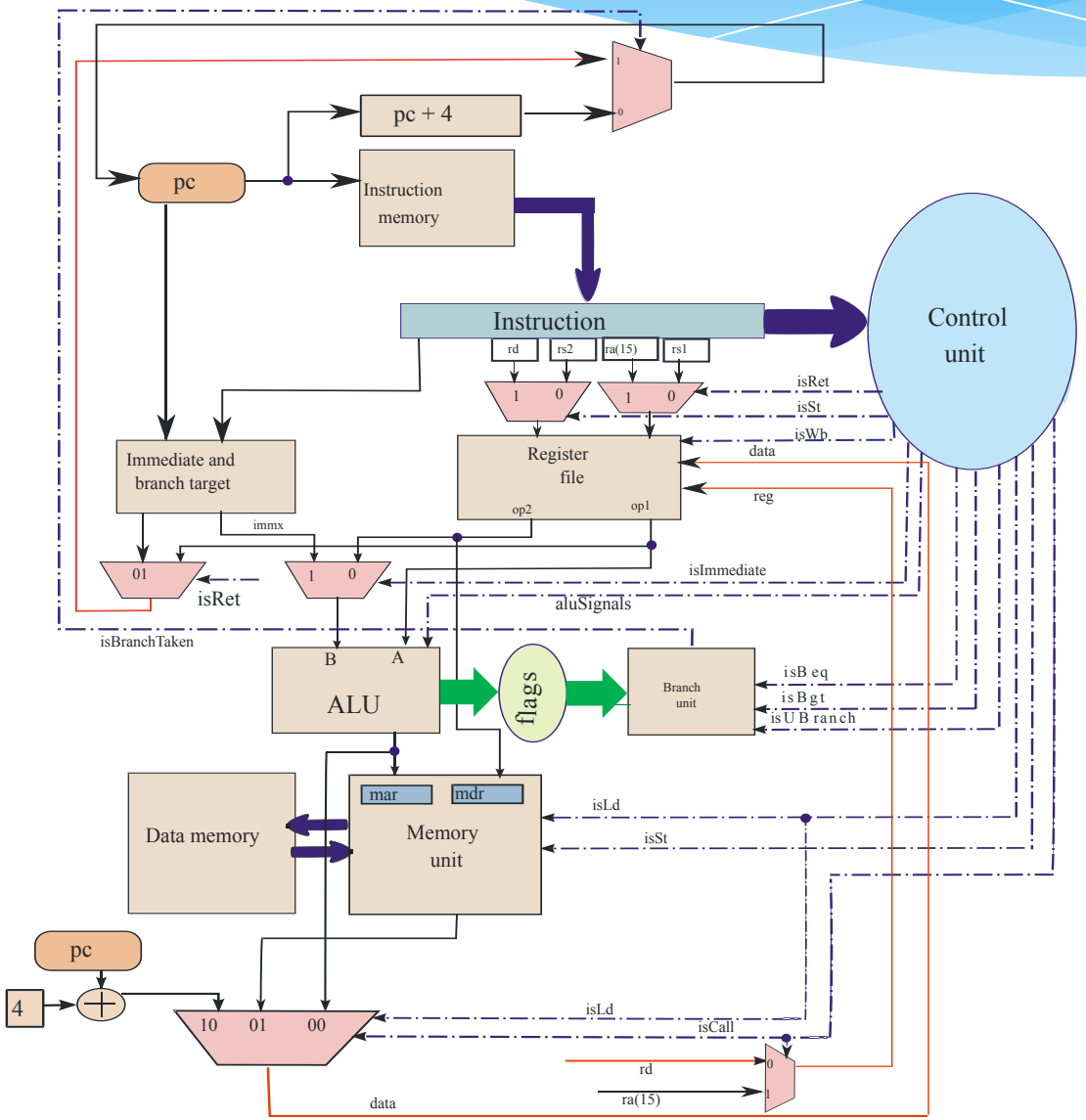

Execute unit
branchP
C
from OF
branchTarget
op1
0
1
isRet
isBranchTa
ken
isUBranch
isBeq
flags.E
isBgt
flags.GT
flags
ALU and mem insts
op1
A
B
imm
x
op2
1
0
ALU
(Arithmetic
logic unit)
aluResult
isImmedi
ate
aluSignal
s

MA Unit


isLd
isSt
32
op2
mdr
Memory unit
32
ldResult
32
aluResult
mar
Data memory
mdr
memory
data reg.
mar
memory
address reg.

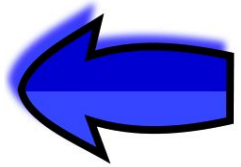
RW Unit


register writeback unit
Register file
A
read port 1
D
A
read port 2
D
write port
E
A
D
isWb
ra(15)
1
0
rd (inst[23:26])
isCall
32
aluResult
32
ldResult
32
pc
4
00
01
10
result
isLd
isCall
E enable
A address
D data


pc + 4
pc
Instruction memory
Instruction
Control unit
rd
rs2
ra(15)
rs1
isRet
isSt
isWb
data
Immediate and branch target
Register file
op2
op1
reg
immx
01
isRet
isImmediate
aluSignals
isBranchTaken
B
A
ALU
flags
Branch unit
isB eq
isB gt
isU B ranch
Data memory
mar
mdr
Memory unit
isLd
isSt
pc
4
10
01
00
isLd
isCall
rd
ra(15)
data

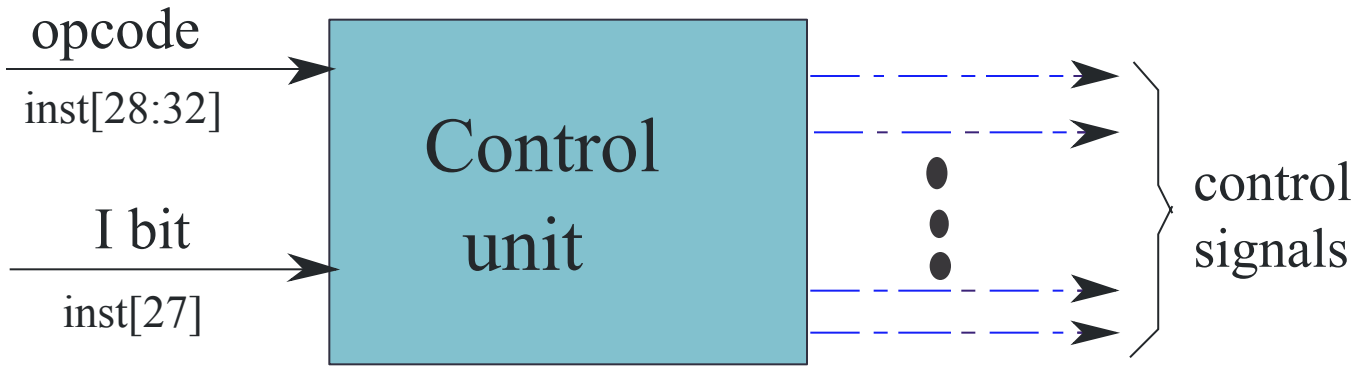
Outline

* Outline of a Processor
* Detailed Design of each Stage
* The Control Unit
* Microprogrammed Processor
* Microassembly Language
* The Microcontrol Unit



Mc Graw Hill Education

The Ha rdwired Control Unit



* Given the opcode and the immediate bit
  * It generates all the control signals

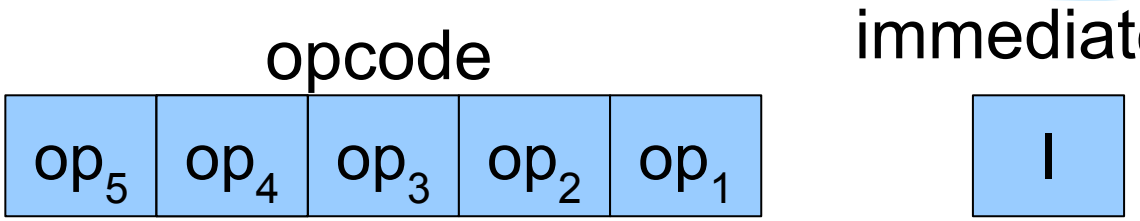
Control Signals

SerialNo.	Signal	Condition
1	*isSt*	Instruction: *st*
2	*isLd*	Instruction: *ld*
3	*isBeq*	Instruction: *beq*
4	*isBgt*	Instruction: *bgt*
5	*isRet*	Instruction: *ret*
6	*isImmediate*	*I* bit set to 1
7	*isWb*	Instructions: *add*, *sub*, *mul*, *div*, *mod*, *and*, *or*, *not*, *mov*, *ld*, *lsl*, *lsr*, *asr*, *call*
8	*isUBranch*	Instructions: *b*, *call*, *ret*
9	*isCall*	Instructions: *call*

Control Signals – II

	aluSignal	
10	*isAdd*	Instructions: *add*, *ld*, *st*
1	*isSub*	Instruction: *sub*
12	*isCmp*	Instruction: *cmp*
13	*isMul*	Instruction: *mul*
14	*isDiv*	Instruction: *div*
15	*isMod*	Instruction: *mod*
16	*isLsl*	Instruction: *lsl*
17	*isLsr*	Instruction: *lsr*
18	*isAsr*	Instruction: *asr*
19	*isOr*	Instruction:
20	*isAnd*	Instruction:
21	*isNot*	Instruction: *not*
22	*isMov*	Instruction: *mov*

Control signal Logic



Serial No.	Signal	Condition
1	*isSt*	$\overline{op_5}.op_4.op_3.op_2.op_1$
2	*isLd*	$\overline{op_5}.op_4.op_3.op_2.\overline{op_1}$
3	*isBeq*	$op_5.\overline{op_4}.\overline{op_3}.\overline{op_2}.\overline{op_1}$
4	*isBgt*	$op_5.\overline{op_4}.\overline{op_3}.\overline{op_2}.op_1$
5	*isRet*	$op_5.\overline{op_4}.op_3.\overline{op_2}.\overline{op_1}$
6	*isImmediate*	I
7	*isWb*	$\sim(op_5 + \overline{op_5}.op_3.op_1.(op_4 + \overline{op_2})) + op_5.op_4.op_3.op_2.op_1$
8	*isUbranch*	$op_5.\overline{op_4}.(\overline{op_3}.op_2 + op_3.\overline{op_2}.\overline{op_1})$
9	*isCall*	$op_5.\overline{op_4}.\overline{op_3}.op_2.op_1$

Control Signal Logic - II

aluSignals		
10	*isAdd*	$\overline{op5}.\overline{op4}.\overline{op3}.\overline{op2}.\overline{op1} + \overline{op5}.op4.op3.op2$
11	*isSub*	$\overline{op5}.\overline{op4}.\overline{op3}.\overline{op2}.op1$
12	*isCmp*	$\overline{op5}.\overline{op4}.op3.\overline{op2}.op1$
13	*isMul*	$\overline{op5}.\overline{op4}.\overline{op3}.op2.\overline{op1}$
14	*isDiv*	$\overline{op5}.\overline{op4}.\overline{op3}.op2.op1$
15	*isMod*	$\overline{op5}.\overline{op4}.op3.\overline{op2}.\overline{op1}$
16	*isLsl*	$\overline{op5}.op4.\overline{op3}.op2.\overline{op1}$
17	*isLsr*	$\overline{op5}.op4.\overline{op3}.op2.op1$
18	*isAsr*	$\overline{op5}.op4.op3.\overline{op2}.\overline{op1}$
19	*isOr*	$\overline{op5}.\overline{op4}.op3.op2.op1$
20	*isAnd*	$\overline{op5}.op4.op3.op2.\overline{op1}$
21	*isNot*	$\overline{op5}.op4.\overline{op3}.\overline{op2}.\overline{op1}$
22	*isMov*	$\overline{op5}.op4.\overline{op3}.\overline{op2}.op1$

Processor State

Memory	
Address	**Contents**
...	
...	
0x00000010	0x00004880
0x00000014	0x00020caf
...	

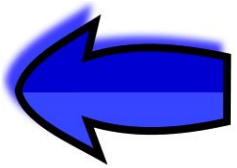
Register File	
Name	**Contents**
r0	0x00000010
r1	0x00000020
r2	0x00000030
r3	0x00000100
...	

Flags Reg	Eq=0; GT=0

PC	0x00000010

Outline

* Outline of a Processor
* Detailed Design of each Stage
* The Control Unit
* Microprogrammed Processor
* Microassembly Language
* The Microcontrol Unit



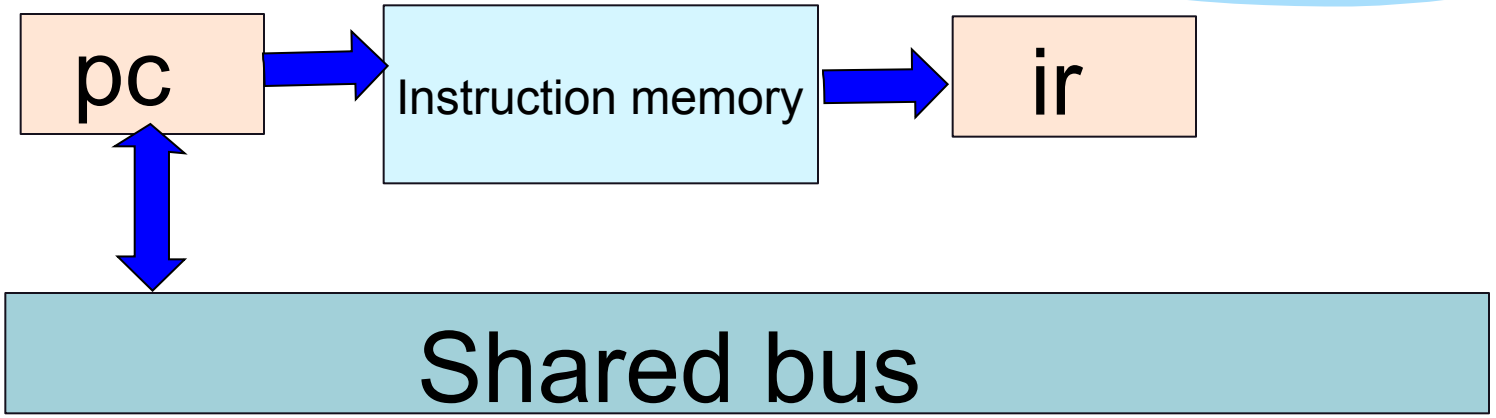
Microprogramming

* Idea of microprogramming
  * Expose the elements in a processor to software
  * Implement instructions as dedicated software routines
* Why make the implementation of instructions flexible ?
  * Dynamically change their behaviour
  * Fix bugs in implementations
  * Implement very complex instructions

Microprogrammed Data Path

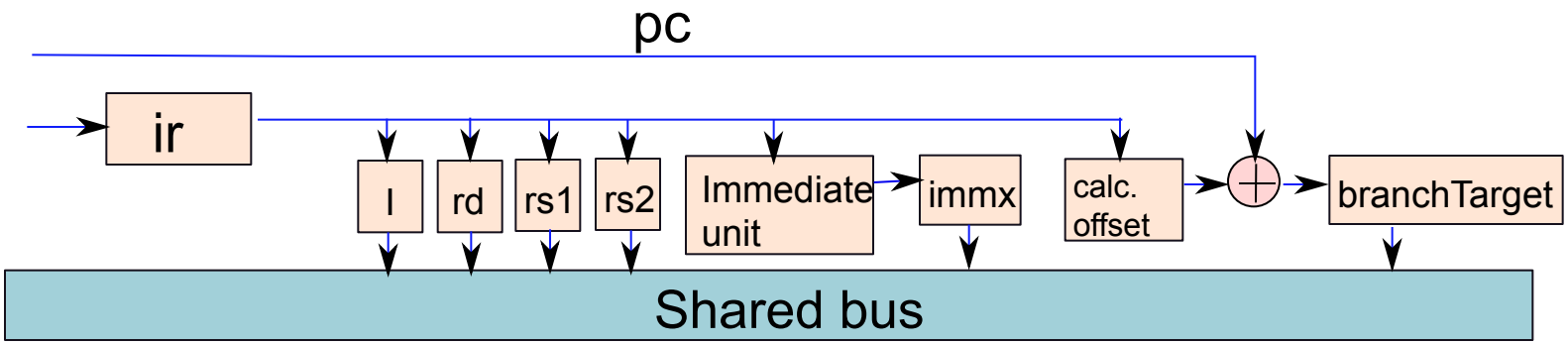
* Expose all the state elements to dedicated system software – firmware
* Write dedicated routines in firmware for implementing each instruction
* Basic idea
  * 1 SimpleRisc Instruction ⟶ Several micro instructions
  * Execute each micro instruction
  * We require a microprogram counter, and microinstruction memory

Fetch Unit



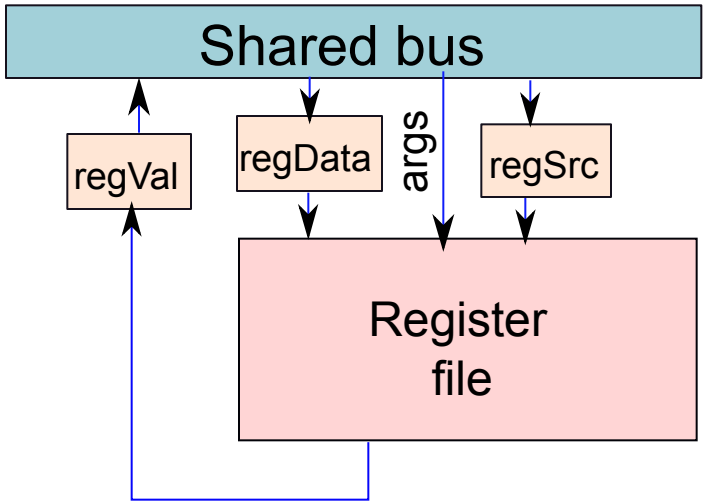
* The pc is used to access the instruction memory.
* The contents of the instruction are saved in the instruction register (ir)

Decode Unit



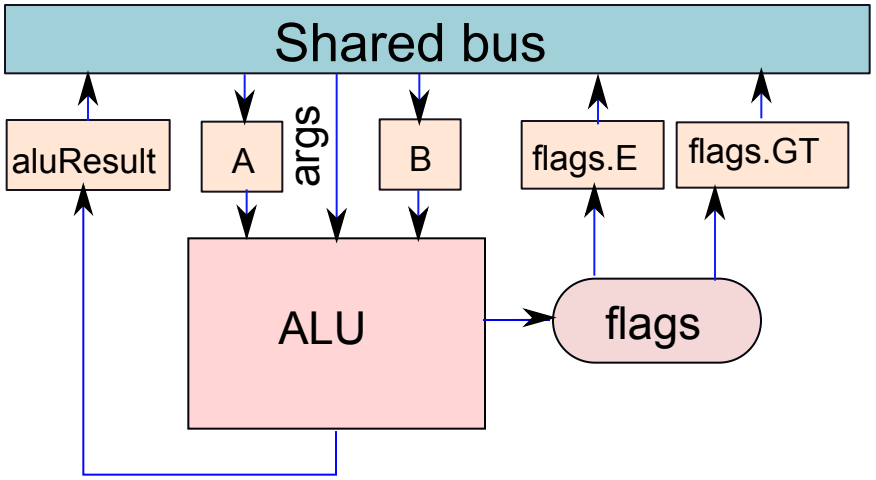
* Divide the contents of ir into different fields
  * I, rd, rs1, rs2, immx, and branchTarget

The Register File



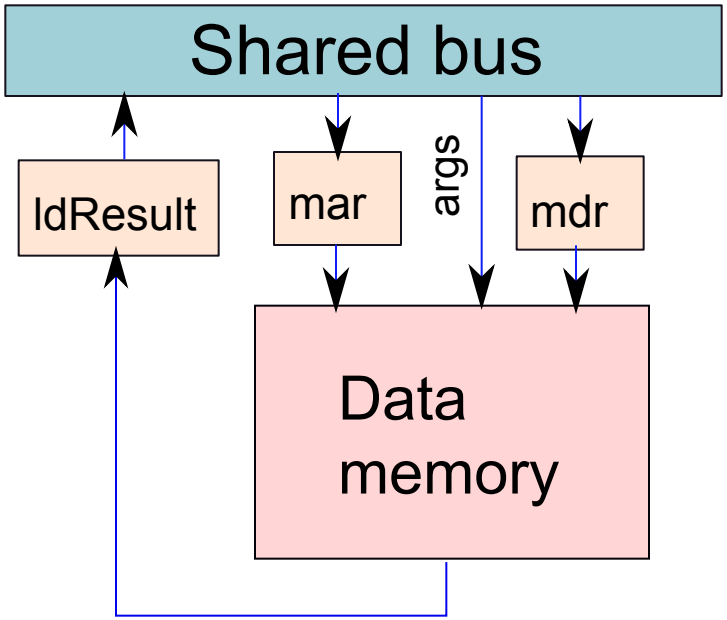
* regSrc (id of the source/dest register)
* regData (data to be stored)
* regVal (register value)

ALU

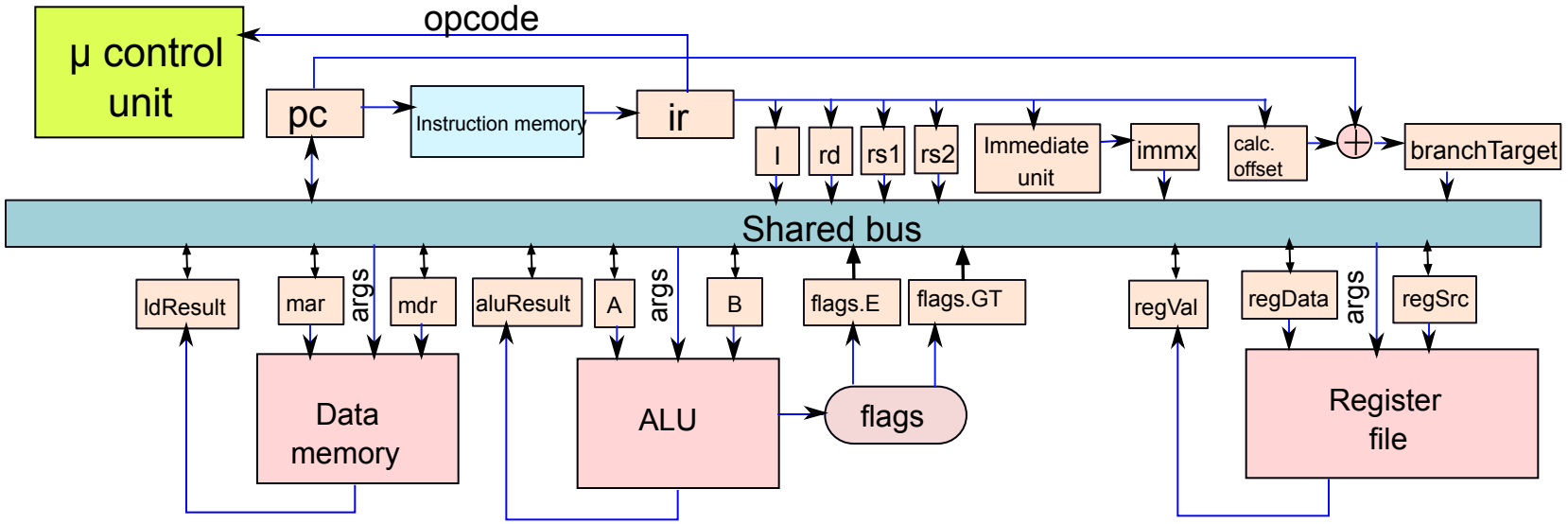


* A, B → ALU operands
* args → ALU operation type
* aluResult → ALU Result

Memory Unit

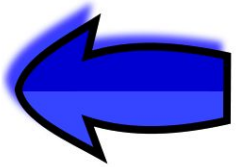


Microprogrammed Data Path



Outline

* Outline of a Processor
* Detailed Design of each Stage
* The Control Unit
* Microprogrammed Processor
* Microassembly Language
* The Microcontrol Unit



Mc Graw Hill Education

Internal Registers

SerialNo.	Register	Size (bits)	Function
1	*pc*	32	program counter
2	*ir*	32	instruction register
3	*I*	1	immediate bit in the instruction
4	*r*	4	destination register id
5	*rs* 1	4	id of the first source register
6	*rs* 2	4	id of the second source register
7	*immx*	32	immediate embedded in the instruction (after processing modifiers)
8	*branchTarget*	32	branch target, computed as the sum of the PC and the offset embedded in the instruction
9	*regSrc*	4	contains the id of the register that needs to be accessed in the register file
10	*regData*	32	contains the data to be written into the register file

Internal Registers - II

11	*regVal*	32	value read from the register file
12	*A*	32	first operand of the AlU
13	*B*	32	second operand of the ALU
14	*flags.E*	1	the equality flag
15	*flags.GT*	1	the greater than flag
16	*aluResult*	32	the ALU result
17	*mar*	32	memory address register
18	*mdr*	32	memory data register
19	*ldResult*	32	the value loaded from memory

Microinstructions

Basic Instructions

* mloadIR → Loads the instruction register (ir) with the contents of the instruction.
* mdecode → Waits for 1 cycle. Meanwhile, all the decode registers get populated
* mswitch → Loads the set of micro instructions corresponding to a program instruction.

Move Microinstructions

* mmov r1, r2 : r1 ← r2
* mmov r1, r2, <args> : r1 ← r2, send the value of args on the bus
* mmovi r1, <imm> : r1 ← imm

Add and Branch Microinstructions

* madd r1, imm, <args>
  * r1 ← r1 + imm
  * send <args> on the bus
* mbeq r1, imm, <label>
  * if (r1 == imm), μpc = addr(label)
* mb <label>
  * μpc = addr(label)

Summary of Microinstructions

SerialNo.	Microinstruction	Semantics
1	*mloadIR*	*ir* ← [*pc*]
2	*mdecode*	populate all the decode registers
3	*mswitch*	jump to the *μpc* corresponding to the opcode
4	*mmov reg*1, *reg*2, < *args* >	*reg*1 ← *reg*2, send the value of *args* to the unit that owns *reg*1, < *args* > is optional
5	*mmovi reg*1, *imm*, < *args* >	*reg*1 ← *imm*, < *args* > is optional
6	*madd reg*1, *imm*, < *args* >	*reg*1 ← *reg*1+*imm*, < *args* > is optional
7	*mbeq reg*1, *imm*, < *label* >	if (*reg*1 = *imm*) *μpc* ← *addr*(*label*)
8	*mb* <*label*>	*μpc* ← *addr*(*label*)

Implementing Instructions in Microcode

* The microcode preamble

```
.begin:
mloadIR
mdecode
madd pc, 4
mswitch
```

  * Load the program counter
  * Decode the instruction
  * Add 4 to the pc
  * Switch to the first microinstruction in the microcode sequence of the prog. instruction

3 Address Format ALU Instruction

```
/* transfer the first operand to the ALU */
mmov regSrc, rs1, <read>
mmov A, regVal

/* check the value of the immediate register */
mbeq I, 1, .imm
/* second operand is a register */
mmov regSrc, rs2, <read>
mmov B, regVal, <aluop>
mb .rw
/* second operand is an immediate */
.imm:
mmov B, immx, <aluop>

/* write the ALU result to the register file*/
.rw:
mmov regSrc, rd
mmov regData, aluResult, <write>
mb .begin
```

The *mov* Instruction

mov instruction

```
/* check the value of the immediate register */
mbeq I, 1, .imm
/* second operand is a register */
mmov regSrc, rs2, <read>
mmov regData, regVal
mb .rw
/* second operand is an immediate */
.imm:
mmov regData, immx

/* write to the register file*/
.rw:
mmov regSrc, rd, <write>

/* jump to the beginning */
mb .begin
```

The *not* Instruction

not instruction

```
/* check the value of the immediate register */
mbeq I, 1, .imm
/* second operand is a register */
mmov regSrc, rs2, <read>
mmov B, regVal, <not> /* ALU operation */
mb .rw
/* second operand is an immediate */
.imm:
mmov B, immx, <not> /* ALU operation */

/* write to the register file*/
.rw:
mmov regData, aluResult
mmov regSrc, rd, <write>

/* jump to the beginning */
mb .begin
```

The *cmp* Instruction

cmp instruction

```
/* transfer rs1 to register A */
mov regSrc, rs1, <read>
mov A, regVal

/* check the value of the immediate register
mbeq I, 1, .imm

/* second operand is a register */
mmov regSrc, rs2, <read>
mmov B, regVal, <cmp> /* ALU operation */
mb .begin

/* second operand is an immediate */
.imm:
mmov B, immx, <cmp> /* ALU operation */
mb .begin
```

The *nop* Instruction

* mb .begin

The *ld* Instruction

ld instruction

```
/* transfer rs1 to register A */
mmov regSrc, rs1, <read>
mmov A, regVal

/* calculate the effective address */
mmov B, immx, <add> /* ALU operation */

/* perform the load */
mmov mar, aluResult, <load>

/* write the loaded value to the register file */
mmov regData, ldResult
mmov regSrc, rd, <write>

/* jump to the beginning */
mb .begin
```

The *st* Instruction

st instruction

```
/* transfer rs1 to register A */
mmov regSrc, rs1, <read>
mmov A, regVal

/* calculate the effective address */
mmov B, immx, <add> /* ALU operation */

/* perform the store */
mmov mar, aluResult
mmov regSrc, rd, <read>
mmov mdr, regVal, <store>

/* jump to the beginning */
mb .begin
```

beq and *bgt* Instructions

beq instruction

```
/* test the flags register
mbeq flags.E, 1, .branch
mb .begin

.branch:
mmov pc, branchTarget
mb .begin
```

bgt instruction

```
/* test the flags register
 mbeq flags.GT, 1, .branch
 mb .begin

 .branch:
 mmov pc, branchTarget
 mb .begin
```

call Instruction

call instruction

```
/* save PC + 4 in the return address register */
mmov regData, pc
mmovi regSrc, 15, <write>

/* branch to the function */
mmov pc, branchTarget
mb .begin
```

ret Instruction

ret instruction

```
/* save the contents of the return
    address register in the PC */


mmovi regSrc, 15, <read>
mmov pc, regVal
mb .begin
```

Example

Change the call instruction to store the return address on the stack. The preamble need not be shown.

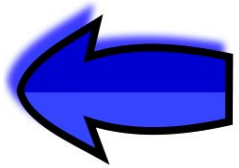
Answer:

stack based call instruction

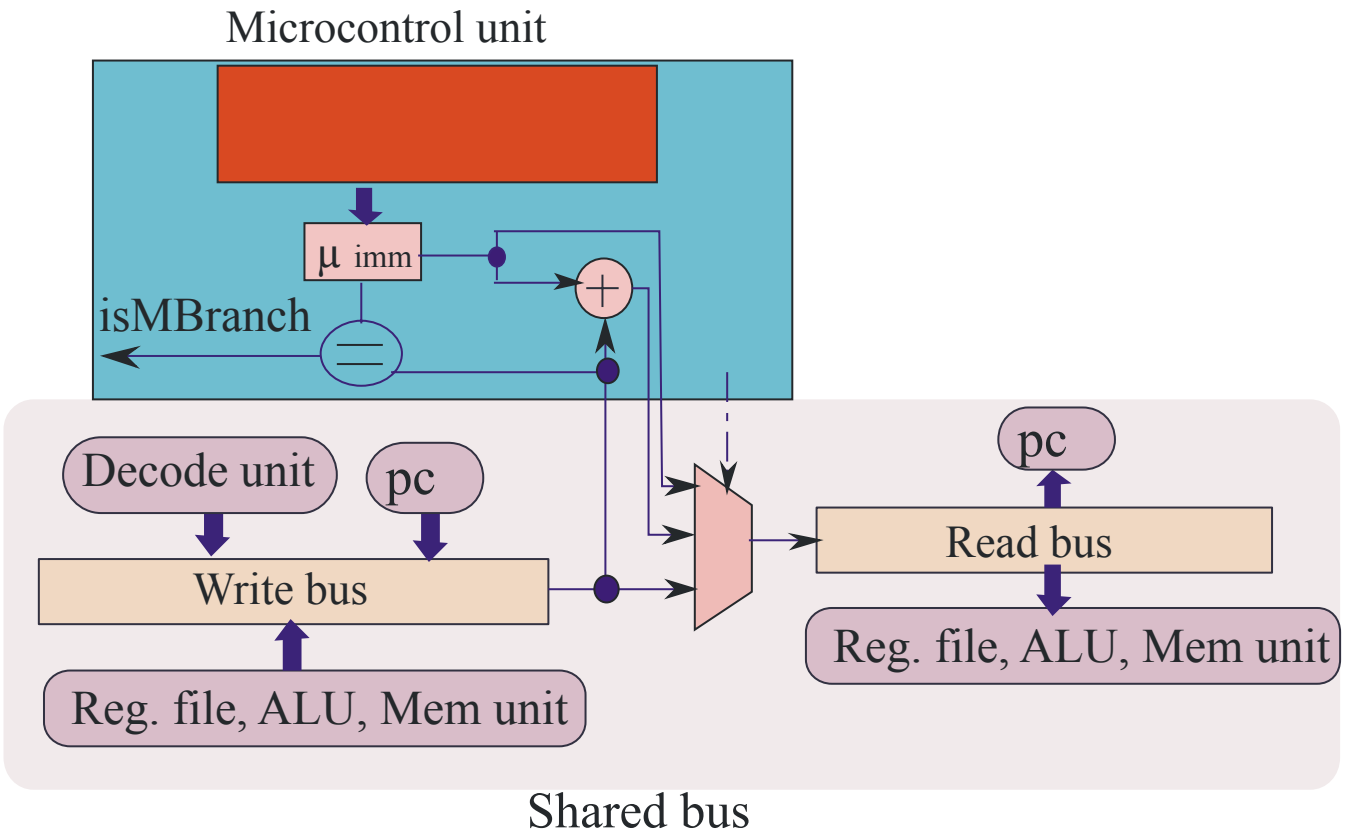
```
/* read the stack pointer */
mmovi regSrc, 14, <read>
madd regVal, -4  /* decrement the stack pointer */

/* set the memory address to the stack pointer */
mmov mar, regVal

/* update the stack pointer */
mmov regData, regVal, <write> /* update stack pointer */

/* write the return address to the stack */
mmov mdr, pc, <store>

/* jump to the beginning */
mb .begin
```

Outline

* Outline of a Processor
* Detailed Design of each Stage
* The Control Unit
* Microprogrammed Processor
* Microassembly Language
* The Microcontrol Unit



Shared Bus


Microcontrol unit
μ imm
isMBranch
Decode unit
pc
Write bus
Reg. file, ALU, Mem unit
pc
Read bus
Reg. file, ALU, Mem unit
Shared bus

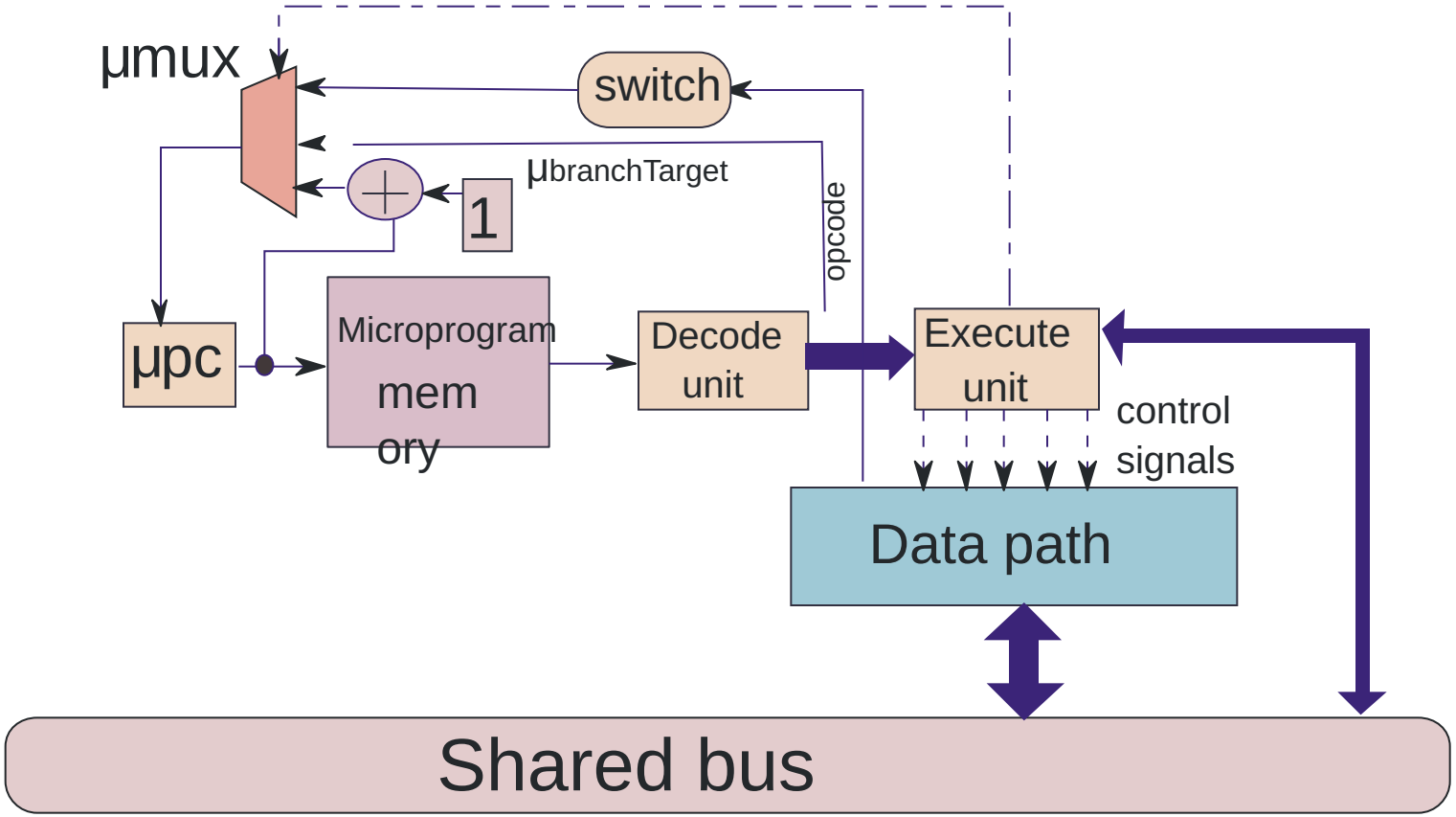
Encoding an Instruction

* Vertical Microprogramming (45 bit inst.)
  * 3 bits → type of instruction
  * 5 bits → source register
  * 5 bits → destination register
  * 12 bits → immediate
  * 10 bit → branch target in microcode memory
  * 10 bit → args value
    * 3 bits → (unit id)
    * 7 bits → operation code

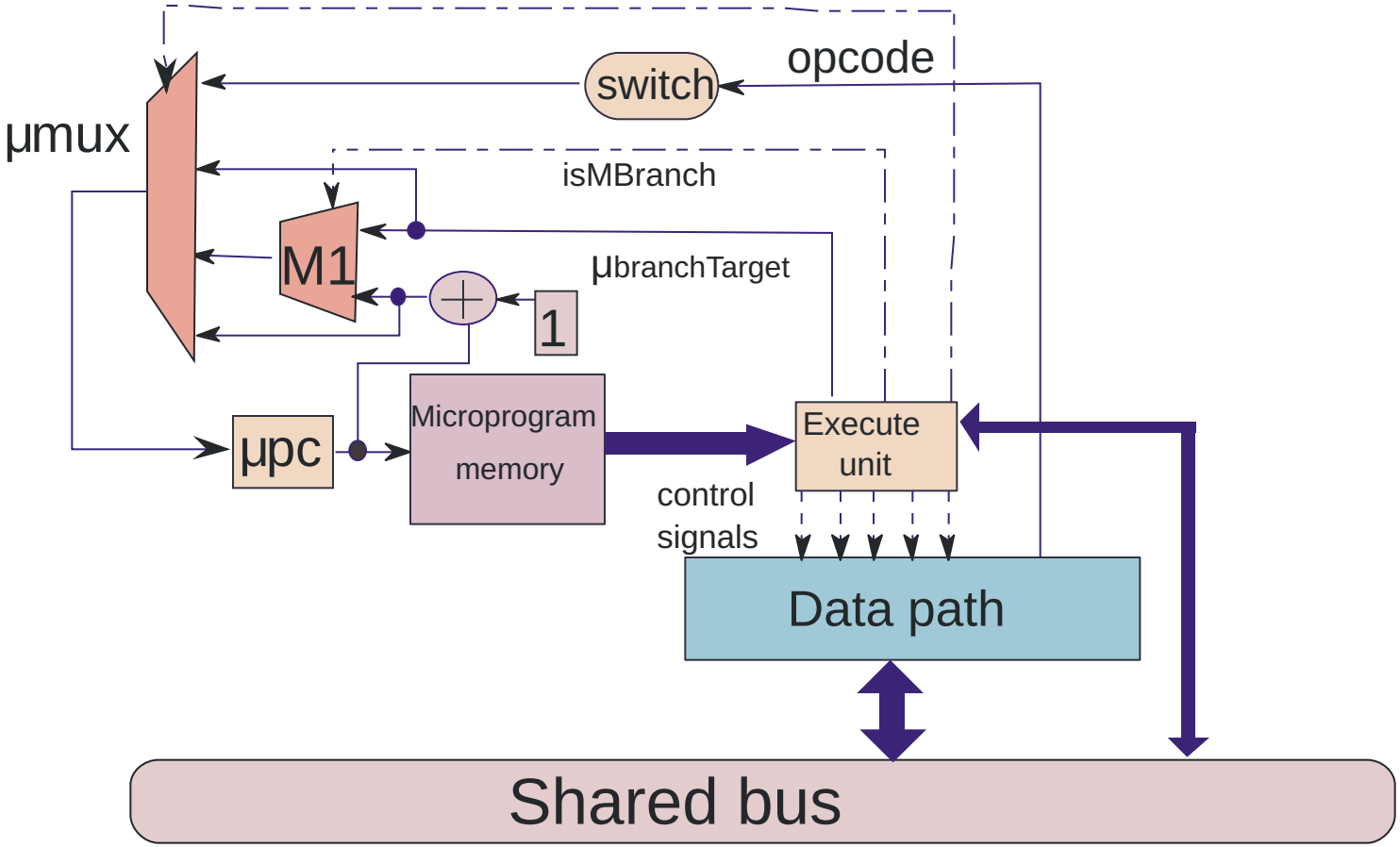
Horizontal Microprogramming

* Encoding
  * 10 bits → branch target
  * 12 bits → immediate
  * 10 bits → args
  * 33 bits → bit vector of all the control signals
* Total size of the encoded instruction : 65 bits

Vertical Microprogramming



Horizontal Microprogramming



THE END