CS 61C: Great Ideas in Computer Architecture

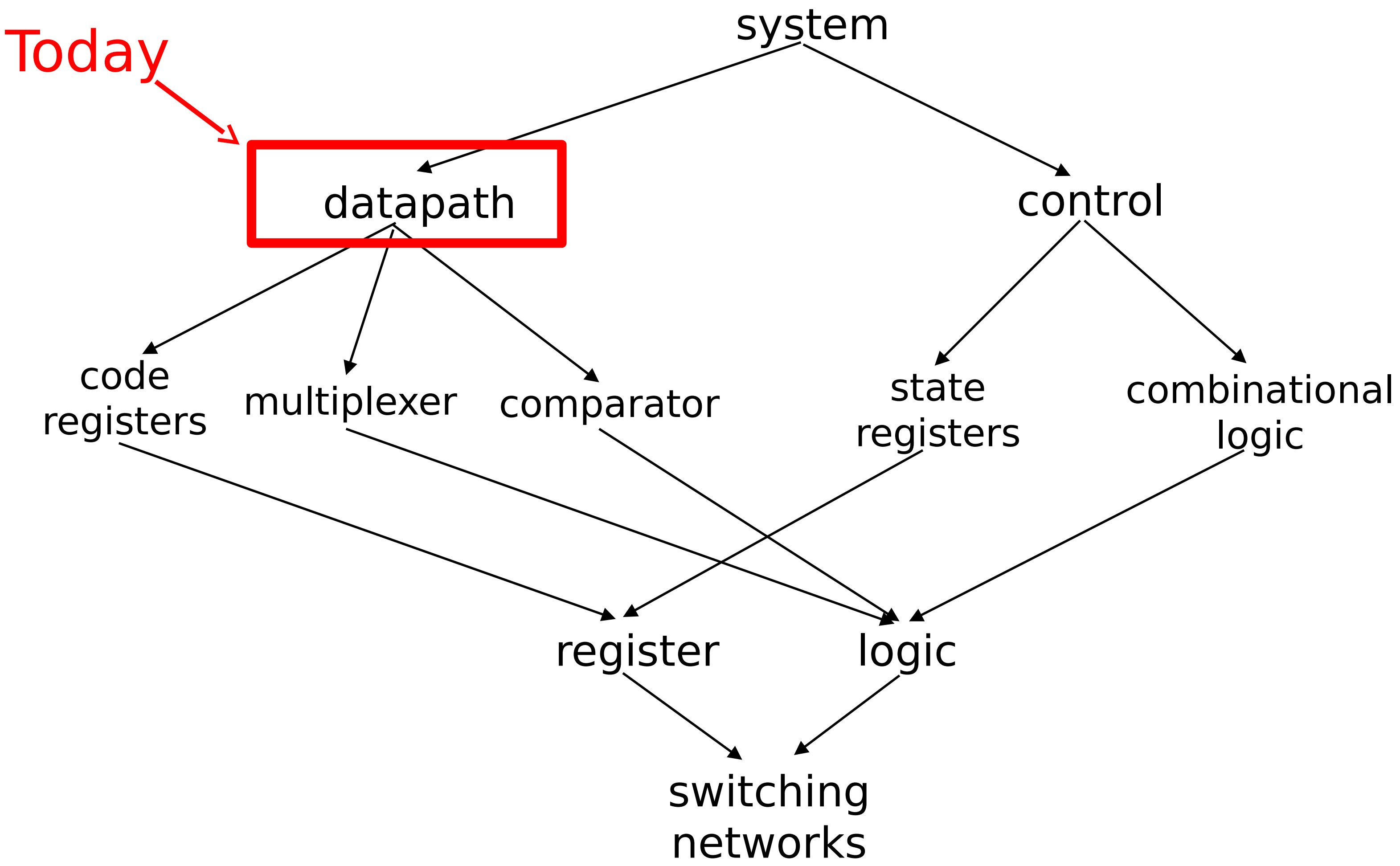
MIPS CPU Datapath, Control Introduction

Instructor: Alan Christopher

Review of Last Lecture

- Critical path constrains clock rate
  - Timing constants: setup, hold, and clk-to-q times
  - Can adjust with extra registers (*pipelining*)
- Finite State Machines examples of sequential logic circuits
  - Can implement systems with Register + CL
- Use MUXes to select among input
  - S input bits selects one of 2^S inputs
- Build n-bit adder out of chained 1-bit adders
  - Can also do subtraction and overflow detection!

Hardware Design Hierarchy

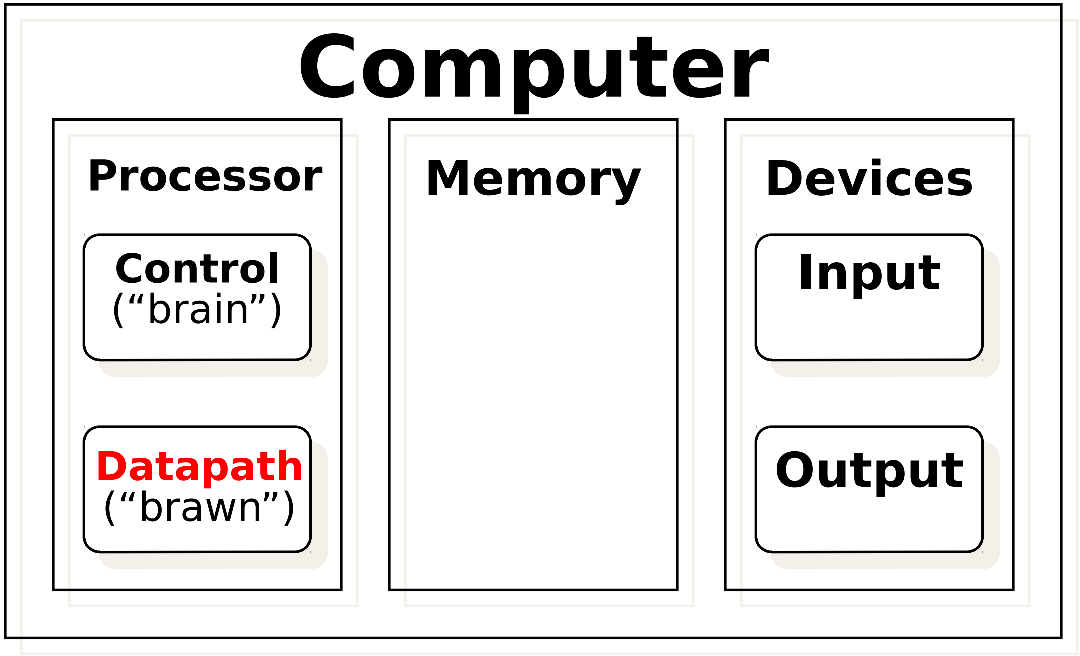


Agenda

- Processor Design
- Administrivia
- Datapath Overview
- Assembling the Datapath
- Control Introduction

Five Components of a Computer

- Components a computer needs to work
  - Control
  - Datapath
  - Memory
  - Input
  - Output



The Processor

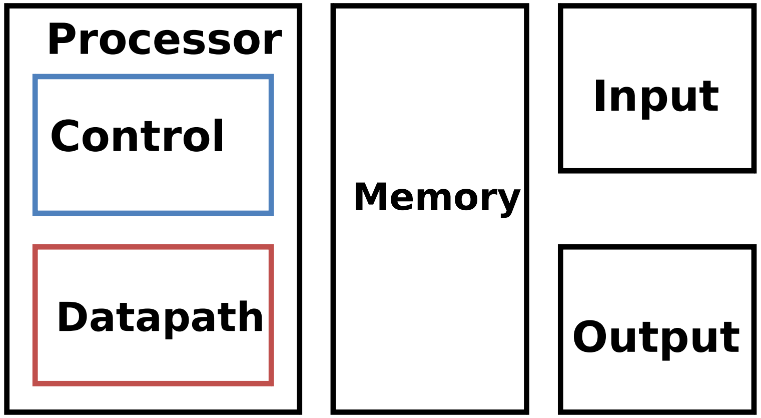
- ***Processor (CPU):*** Implements the instructions of the Instruction Set Architecture (ISA)
  - *Datapath:* part of the processor that contains the hardware necessary to perform operations required by the processor (“the brawn”)
  - *Control:* part of the processor (also in hardware) which tells the datapath what needs to be done (“the brain”)

Processor Design Process

- Five steps to design a processor:

Now

1. Analyze instruction set **->** datapath requirements
2. Select set of datapath components & establish clock methodology
3. Assemble datapath meeting the requirements
4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer
5. Assemble the control logic
    - Formulate Logic Equations
    - Design Circuits



The MIPS-lite Instruction Subset

- ADDU and SUBU
  - `addu rd,rs,rt`
  - `subu rd,rs,rt`

31 26	21	16	11	6	0
op	**rs**	**rt**	**rd**	**shamt**	**funct**
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- OR Immediate:
  - `ori rt,rs,imm16`

31 26	21	16	0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits

- LOAD and STORE Word
  - `lw rt,rs,imm16`
  - `sw rt,rs,imm16`

31 26	21	16	0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits

- BRANCH:
  - `beq rs,rt,imm16`

31 26	21	16	0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits

Register Transfer Language (RTL)

- All start by *fetching* the instruction:

```
R-format:  {op, rs, rt, rd, shamt, funct} ← MEM[ PC ]
I-format:  {op, rs, rt, imm16} ← MEM[ PC ]
```

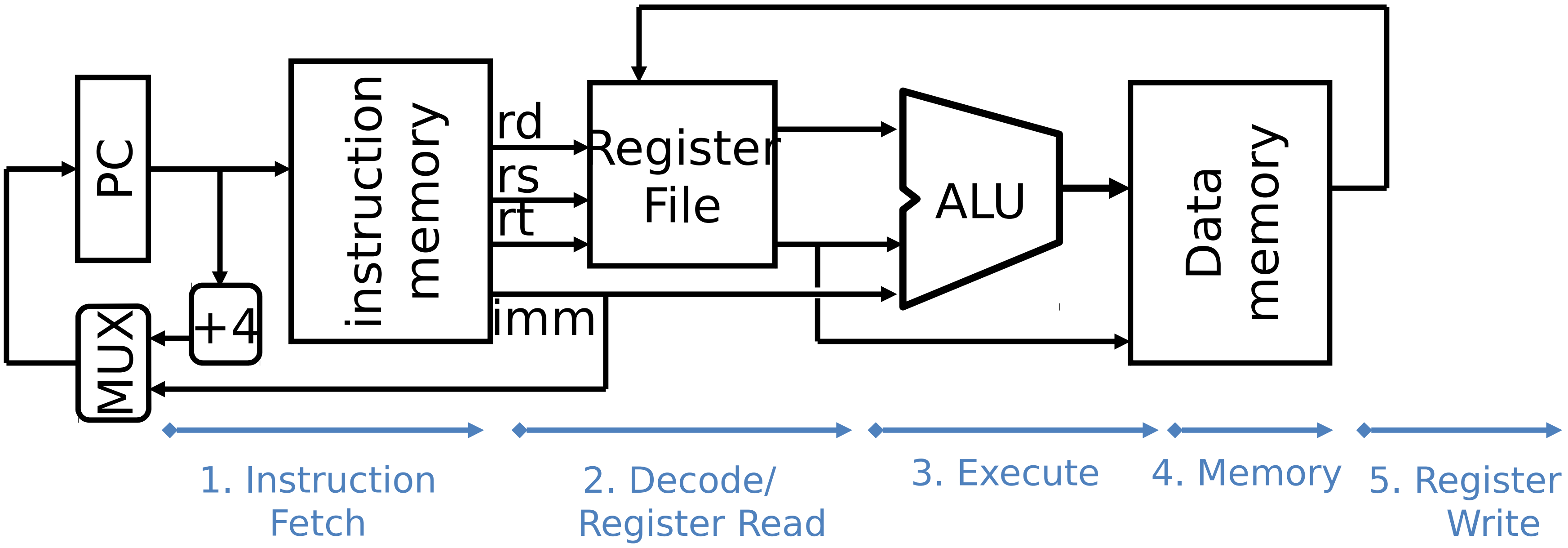
- RTL gives the meaning of the instructions:

```
Inst   Register Transfers
ADDU   R[rd]←R[rs]+R[rt]; PC←PC+4
SUBU   R[rd]←R[rs]–R[rt]; PC←PC+4
ORI    R[rt]←R[rs]|zero_ext(imm16); PC←PC+4
LOAD   R[rt]←MEM[R[rs]+sign_ext(imm16)]; PC←PC+4
STORE  MEM[R[rs]+sign_ext(imm16)]←R[rt]; PC←PC+4
BEQ    if ( R[rs] == R[rt] )
           then PC←PC+4 + (sign_ext(imm16) || 00)
           else PC←PC+4
```

Step 1: Requirements of the Instruction Set

- Memory (MEM)
  - Instructions & data (separate: in reality just caches)
  - Load from and store to
- Registers (32 32-bit regs)
  - Read *rs* and *rt*
  - Write *rt* or *rd*
- PC
  - Add 4 (+ maybe extended immediate)
- Extender (sign/zero extend)
- Add/Sub/OR unit for operation on register(s) or extended immediate
  - Compare if registers equal?

Generic Datapath Layout

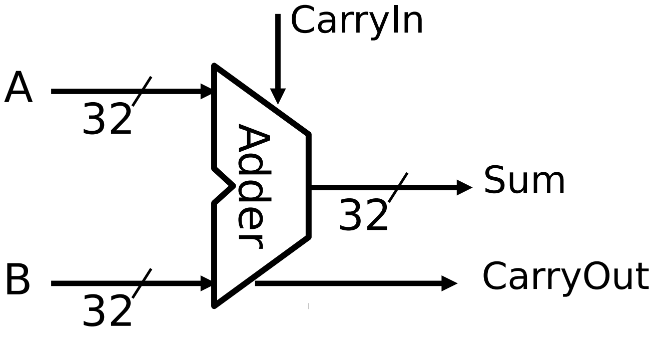


1. Instruction Fetch
2. Decode/ Register Read
3. Execute
4. Memory
5. Register Write

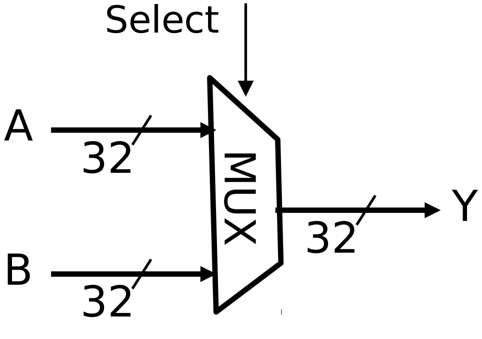
- Break up the process of “executing an instruction”
  - Smaller phases easier to design and modify independently

Step 2: Components of the Datapath

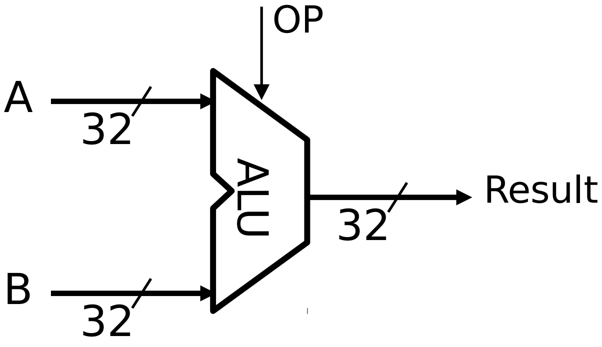
- Combinational Elements
  - Gates and wires
- State Elements + Clock
- Building Blocks:



Adder



Multiplexer



ALU

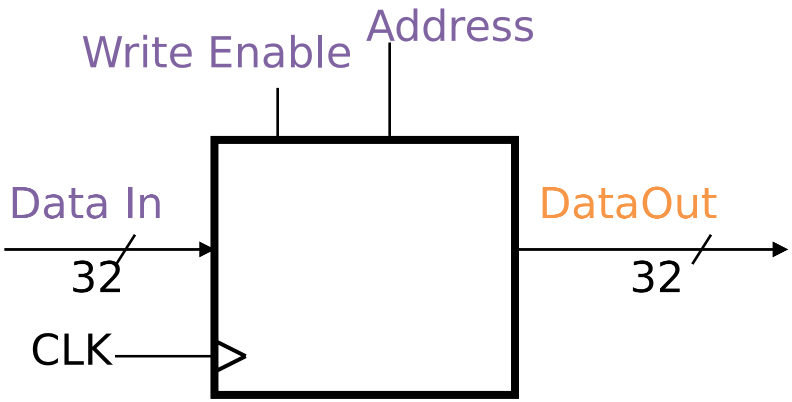
ALU Requirements

- MIPS-lite: add, sub, OR, equality

```
ADDU  R[rd] = R[rs] + R[rt]; ...
SUBU  R[rd] = R[rs] – R[rt]; ...
ORI   R[rt] = R[rs] | zero_ext(Imm16)...
BEQ   if ( R[rs] == R[rt] )
```

- Equality test: Use subtraction and implement output to indicate if result is 0
- P&H also adds AND, Set Less Than (1 if A < B, 0 otherwise)
- Can see ALU from P&H C.5 (on CD)

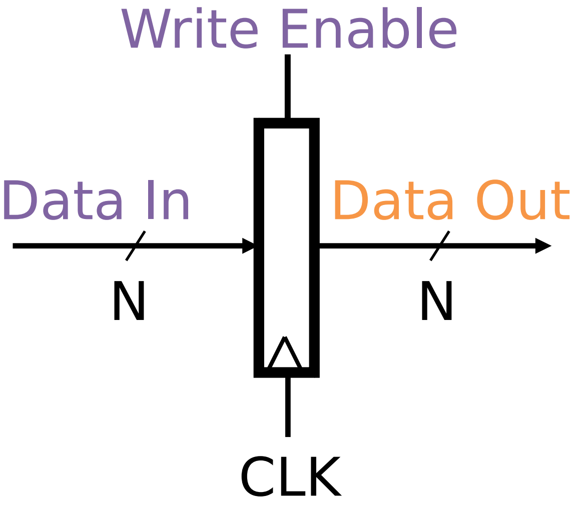
Storage Element: Idealized Memory



- Memory (idealized)
  - One input bus: Data In
  - One output bus: Data Out
- Memory access:
  - Read: Write Enable = 0, data at Address is placed on Data Out
  - Write: Write Enable = 1, Data In written to Address
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read, behaves as a combinational logic block: Address valid => Data Out valid after "access time"

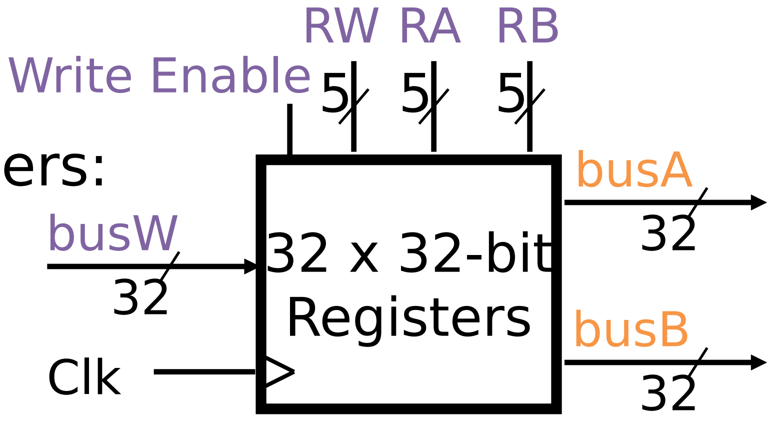
Storage Element: Register

- As seen in Logisim intro
  - N-bit input and output buses
  - Write Enable input



- Write Enable:
  - De-asserted (0): Data Out will not change
  - Asserted (1): Data In value placed onto Data Out after CLK trigger

Storage Element: Register File



- *Register File* consists of 32 registers:
  - Output buses busA and busB
  - Input bus busW
- Register selection
  - Place data of register RA (number) onto busA
  - Place data of register RB (number) onto busB
  - Store data on busW into register RW (number) when Write Enable is 1
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read, behaves as a combinational logic block: RA or RB valid => busA or busB valid after “access time”

Agenda

- Processor Design
- Administrivia
- Datapath Overview
- Assembling the Datapath
- Control Introduction

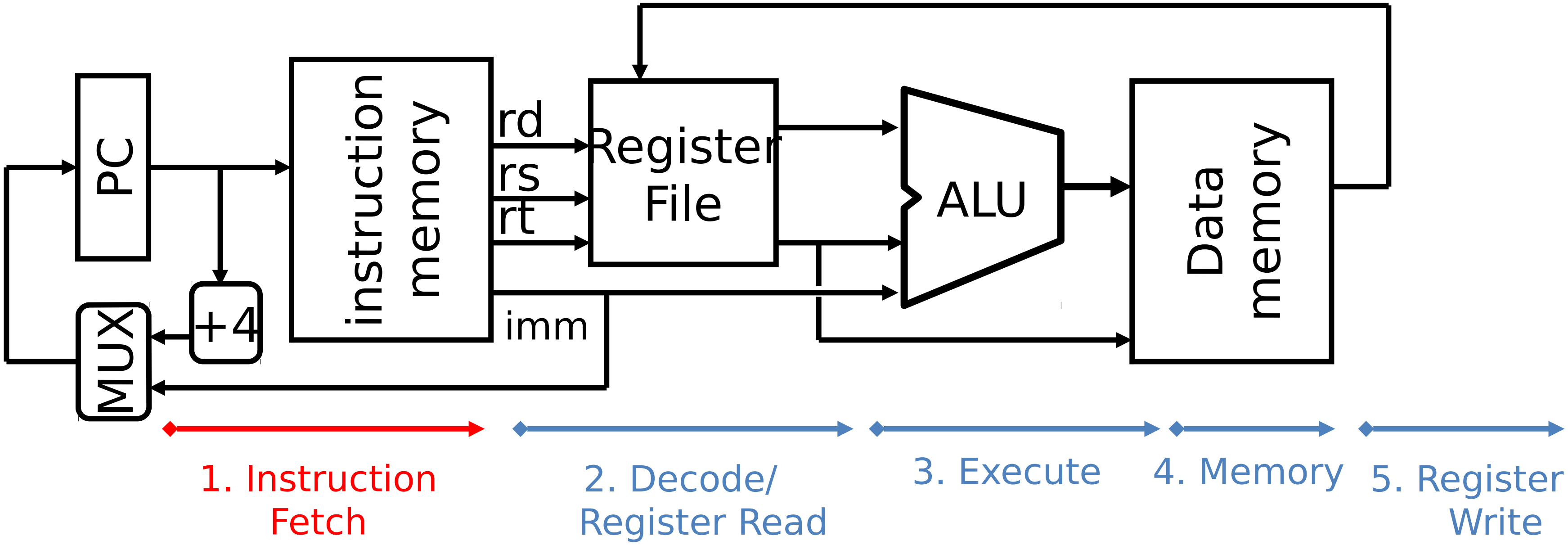
Administrivia

- HW 5 due Thursday
- Project 2 due Sunday
- Project 1 and Midterm graded
  - See piazza for details
- No lab on Thursday – work on HW/project

Agenda

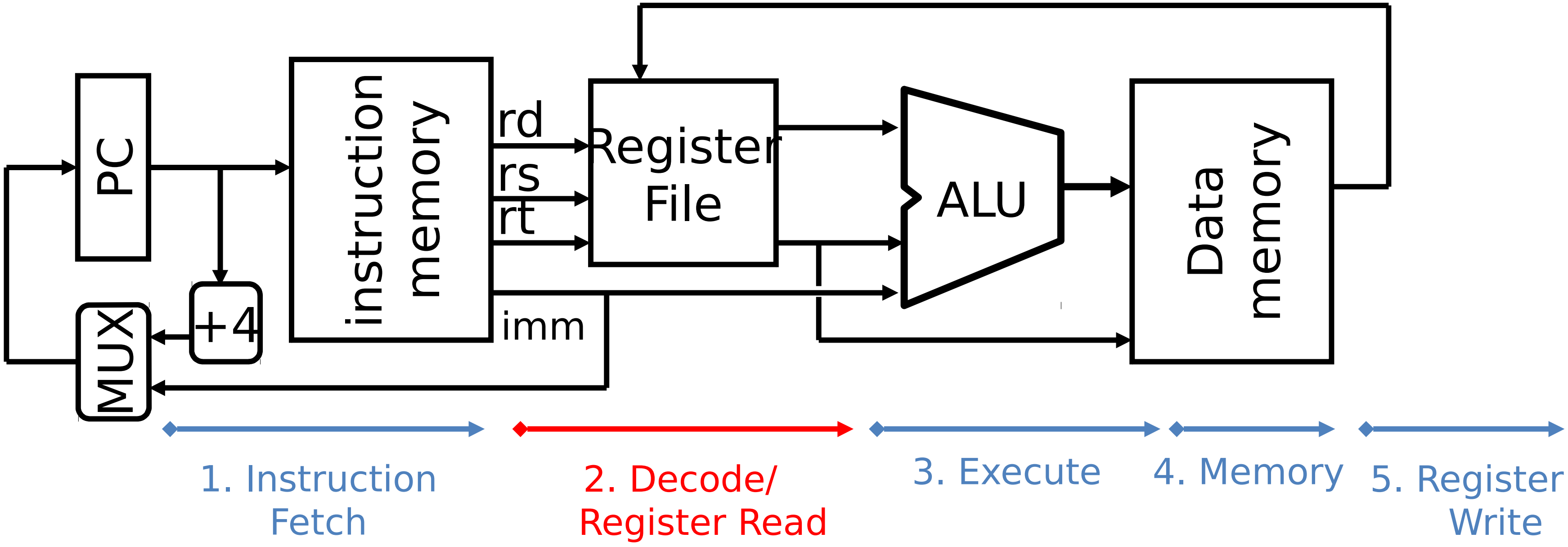
- Processor Design
- Administrivia
- Datapath Overview
- Assembling the Datapath
- Control Introduction
- Clocking Methodology

Datapath Overview (1/5)



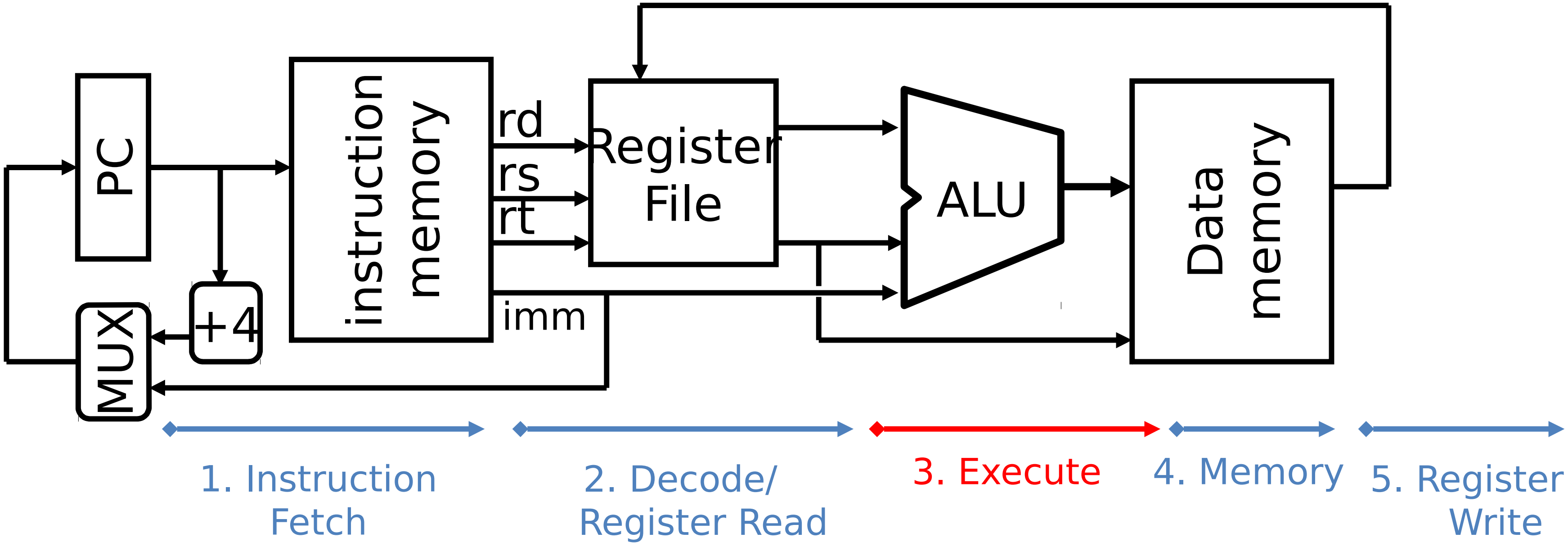
- Phase 1: *Instruction Fetch* (IF)
  - Fetch 32-bit instruction from memory
  - Increment PC (PC = PC + 4)

Datapath Overview (2/5)



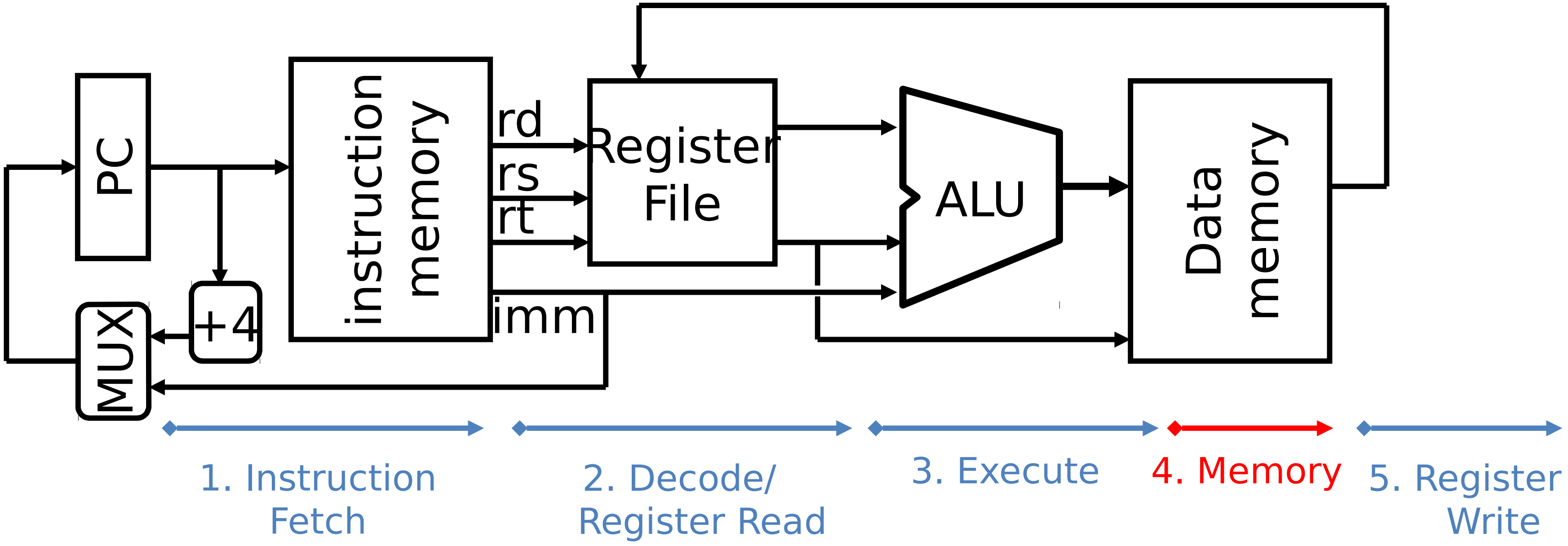
- Phase 2: *Instruction Decode* (ID)
  - Read the opcode and appropriate fields from the instruction
  - Gather all necessary registers values from Register File

Datapath Overview (3/5)



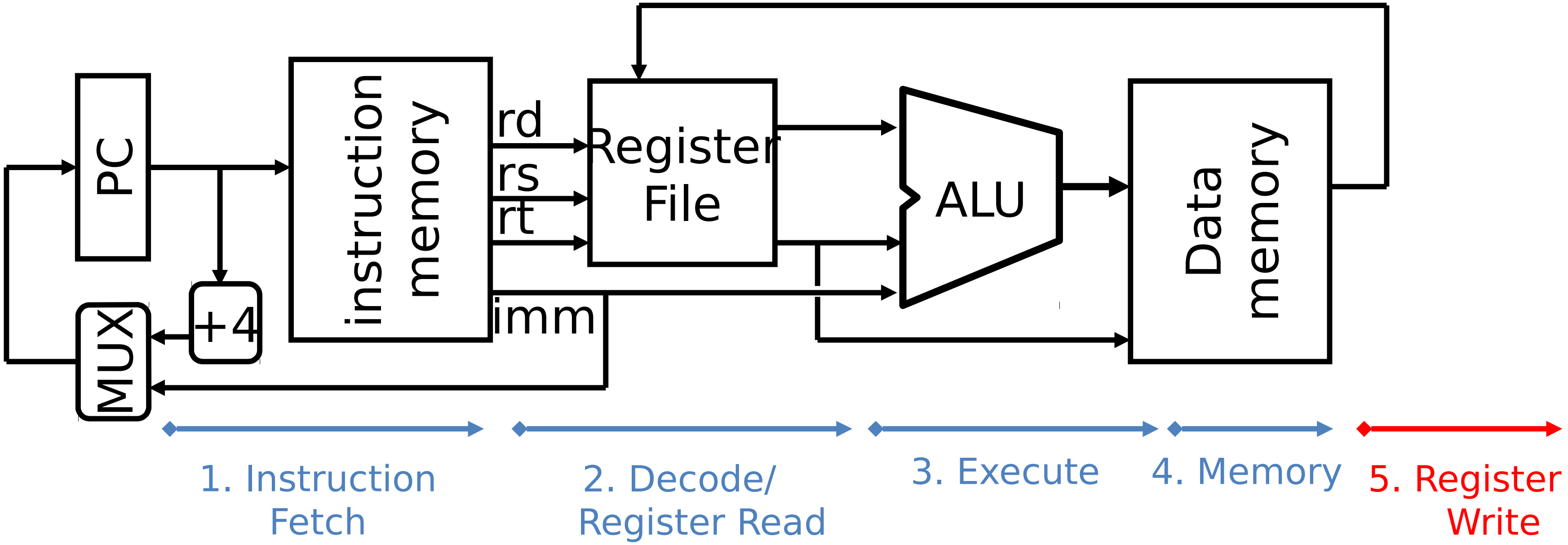
- Phase 3: *Execute* (EX)
  - ALU performs operations: arithmetic (`+`,`-`,`*`,`/`), shifting, logical (`&`,`|`), comparisons (`slt`,`==`)
  - Also calculates addresses for loads and stores

Datapath Overview (4/5)



- Phase 4: *Memory Access* (MEM)
  - Only load and store instructions do anything during this phase; the others remain idle or skip this phase
  - Should hopefully be fast due to caches

Datapath Overview (5/5)



- Phase 5: *Register Write* (WB for "write back")
  - Write the instruction result back into the Register File
  - Those that don't (e.g. `sw`, `j`, `beq`) remain idle or skip this phase

Why Five Stages?

- Could we have a different number of stages?
  - Yes, and other architectures do
- So why does MIPS have five if instructions tend to idle for at least one stage?
  - The five stages are the union of all the operations needed by all the instructions
  - There is one instruction that uses all five stages: *load* (`lw`/`lb`)

Agenda

- Processor Design
- Administrivia
- Datapath Overview
- Assembling the Datapath
- Control Introduction

Step 3: Assembling the Datapath

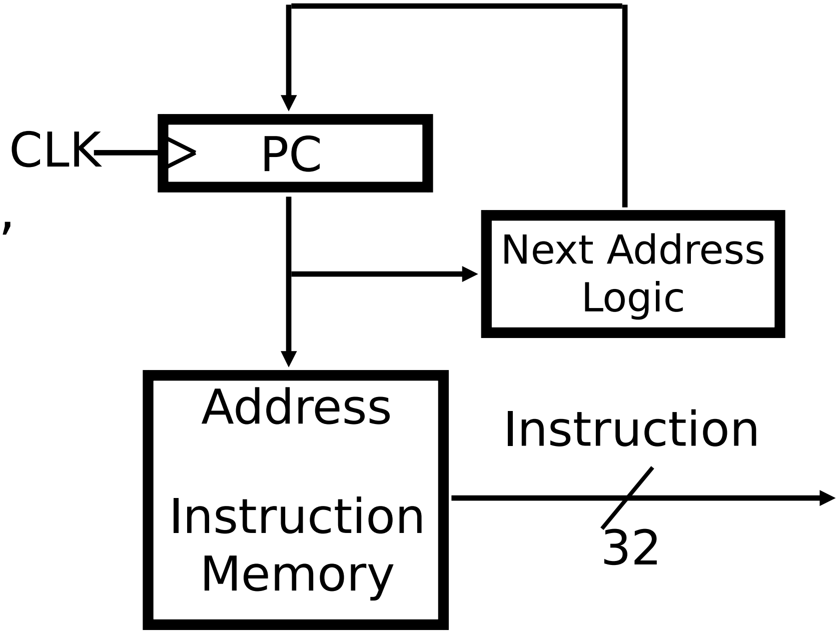
- Assemble datapath to meet RTL requirements
  - Exact requirements will change based on ISA
  - Here we will examine *each instruction* of MIPS-lite
- The datapath is all of the hardware components and wiring necessary to carry out ALL of the different instructions
  - Make sure all components (e.g. RegFile, ALU) have access to all necessary signals and buses
  - Control will make sure instructions are properly executed (the decision making)

Datapath by Instruction

- All instructions: *Instruction Fetch* (**IF**)
  - Fetch the Instruction: Mem[PC]
  - Update the program counter:
    - Sequential Code:

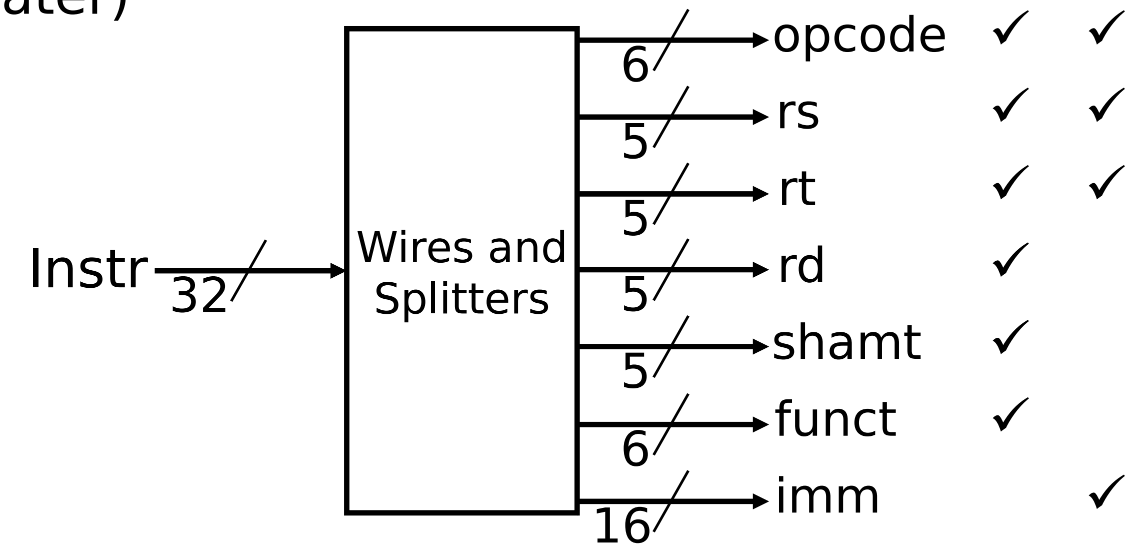
      PC ← PC + 4
    - Branch and Jump:

      PC ← "something else"



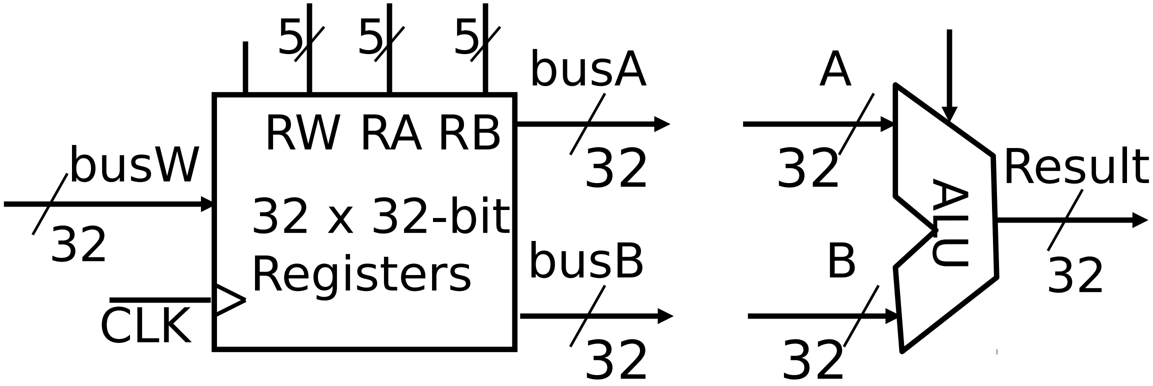
Datapath by Instruction

- All instructions: *Instruction Decode* (**ID**)
  - Pull off all relevant fields from instruction to make available to other parts of datapath
    - MIPS-lite only has **R-format** and **I-format**
    - Control will sort out the proper routing (discussed later)



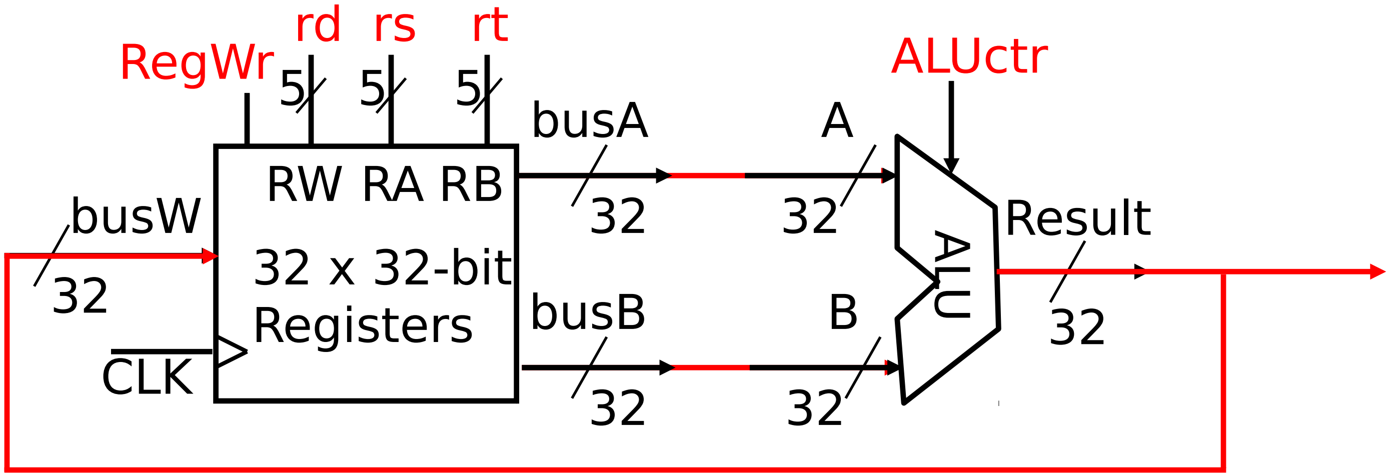
Step 3: Add & Subtract

- `ADDU R[rd]←R[rs]+R[rt];`
- Hardware needed:
  - Instruction Mem and PC (already shown)
  - Register File (RegFile) for read and write
  - ALU for add/subtract



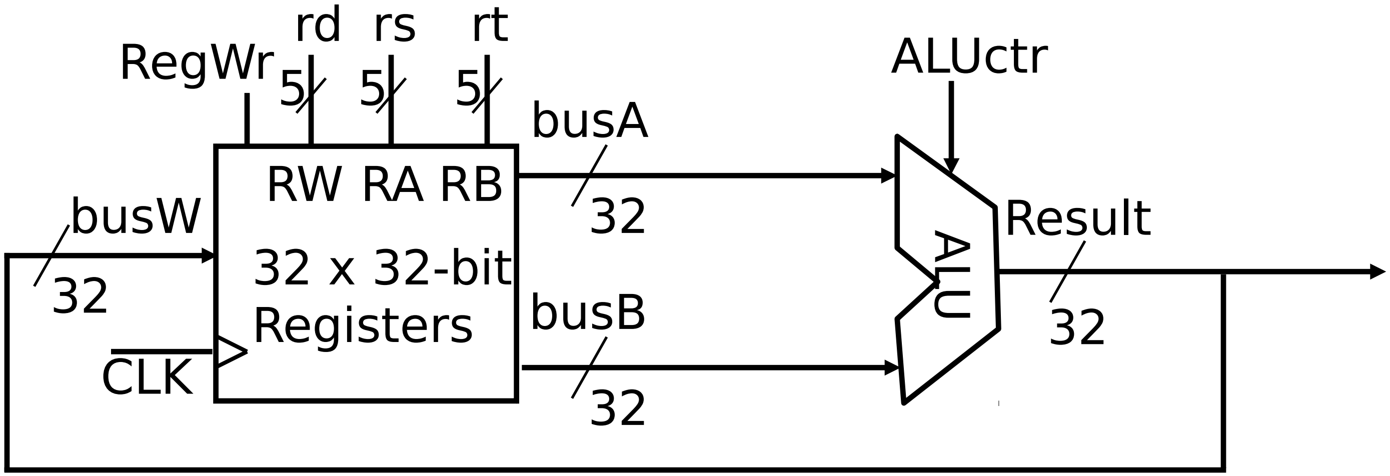
Step 3: Add & Subtract

- `ADDU R[rd]←R[rs]+R[rt];`
- Connections:
  - RegFile and ALU Inputs
  - Connect RegFile and ALU
  - `RegWr` (1) and `ALUctr` (`ADD/SUB`) set by control in **ID**



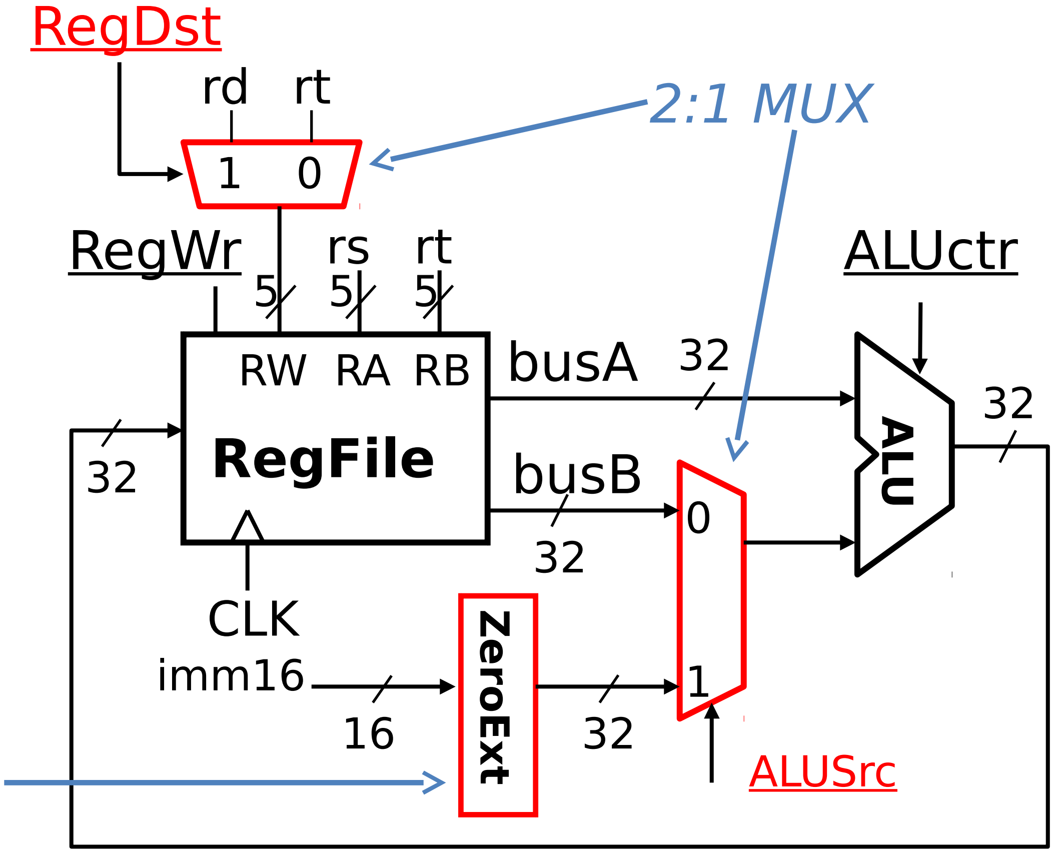
Step 3: Or Immediate

- `ORI R[rt]←R[rs]|zero_ext(Imm16);`
- Is the hardware below sufficient?
  - Zero extend `imm16`?
  - Pass `imm16` to input of ALU?
  - Write result to `rt`?



Step 3: Or Immediate

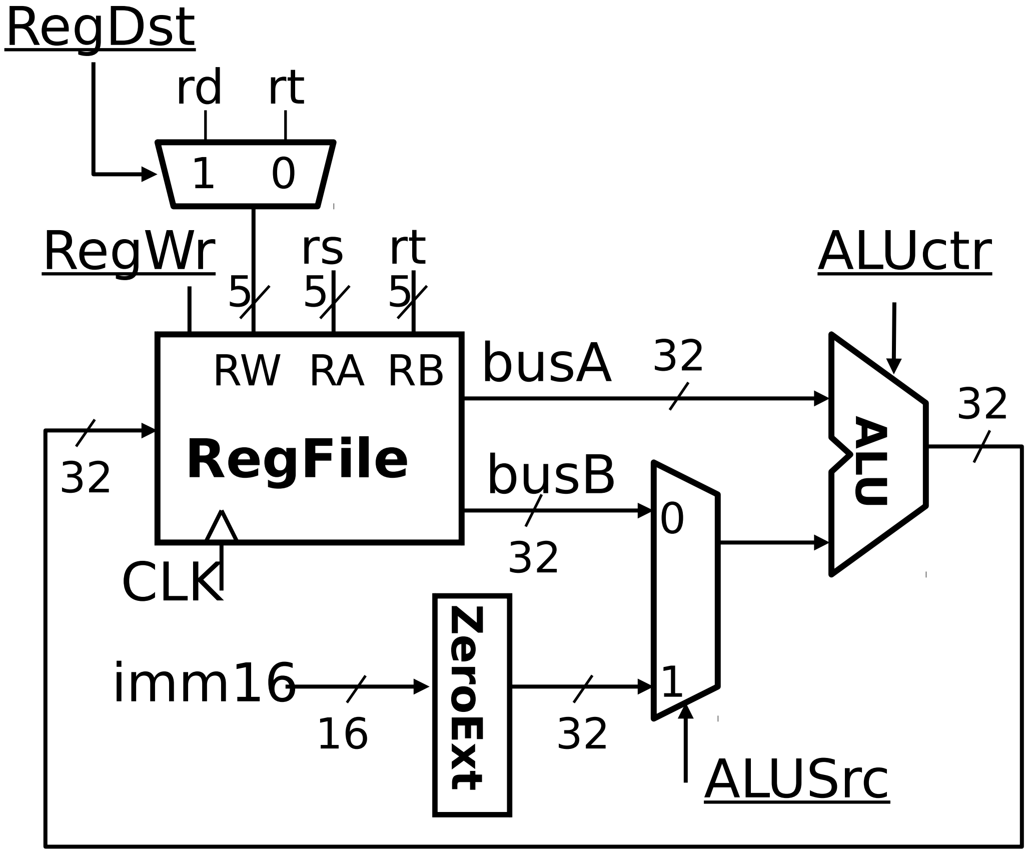
- `ORI R[rt]←R[rs]|zero_ext(Imm16);`
- Add new hardware:
    - Still support add/sub
    - New control signals: `RegDst, ALUSrc`



How to implement this?

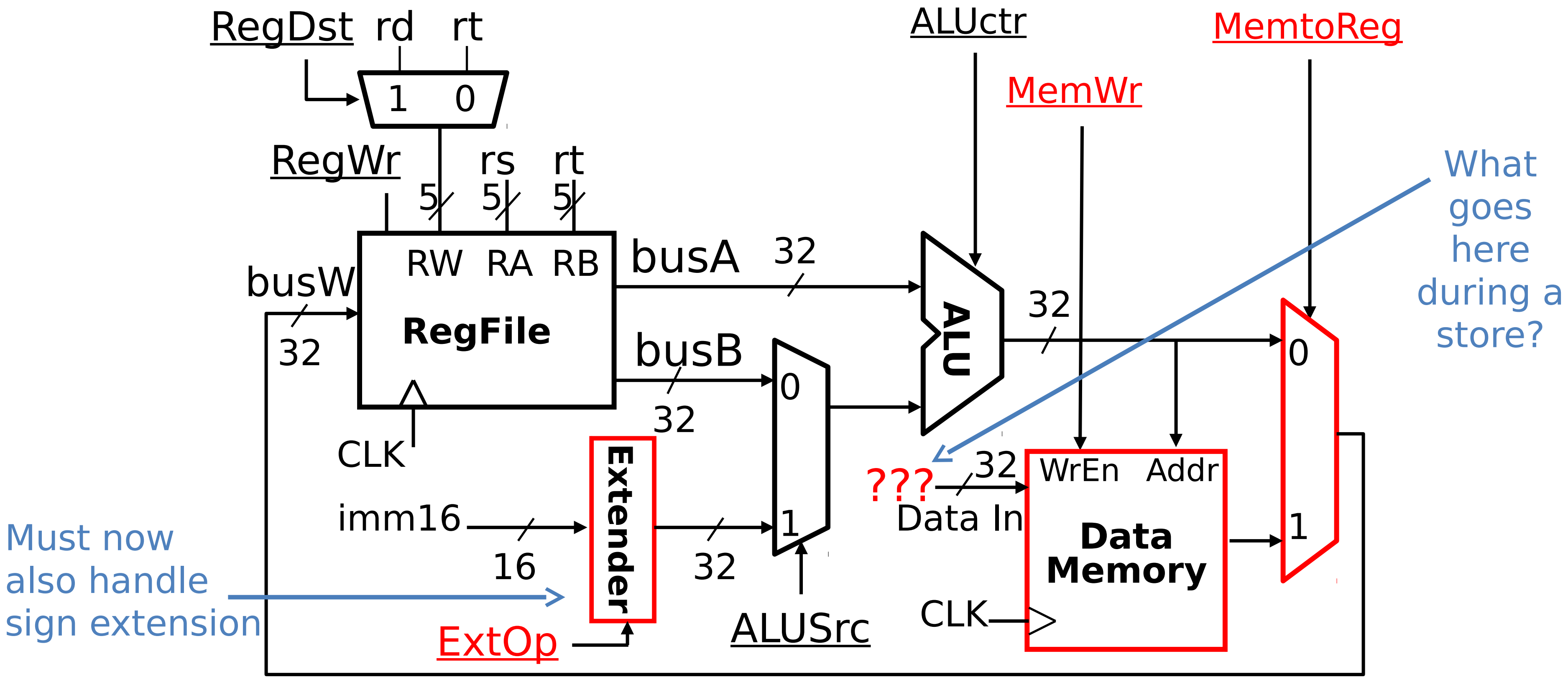
Step 3: Load

- `LOAD R[rt]←MEM[R[rs]+sign_ext(Imm16)];`
- Hardware sufficient?
  - Sign extend `imm16`?
  - Where’s MEM?



Step 3: Load

- `LOAD R[rt]←MEM[R[rs]+sign_ext(Imm16)];`
- New control signals: `ExtOp`, `MemWr`, `MemtoReg`

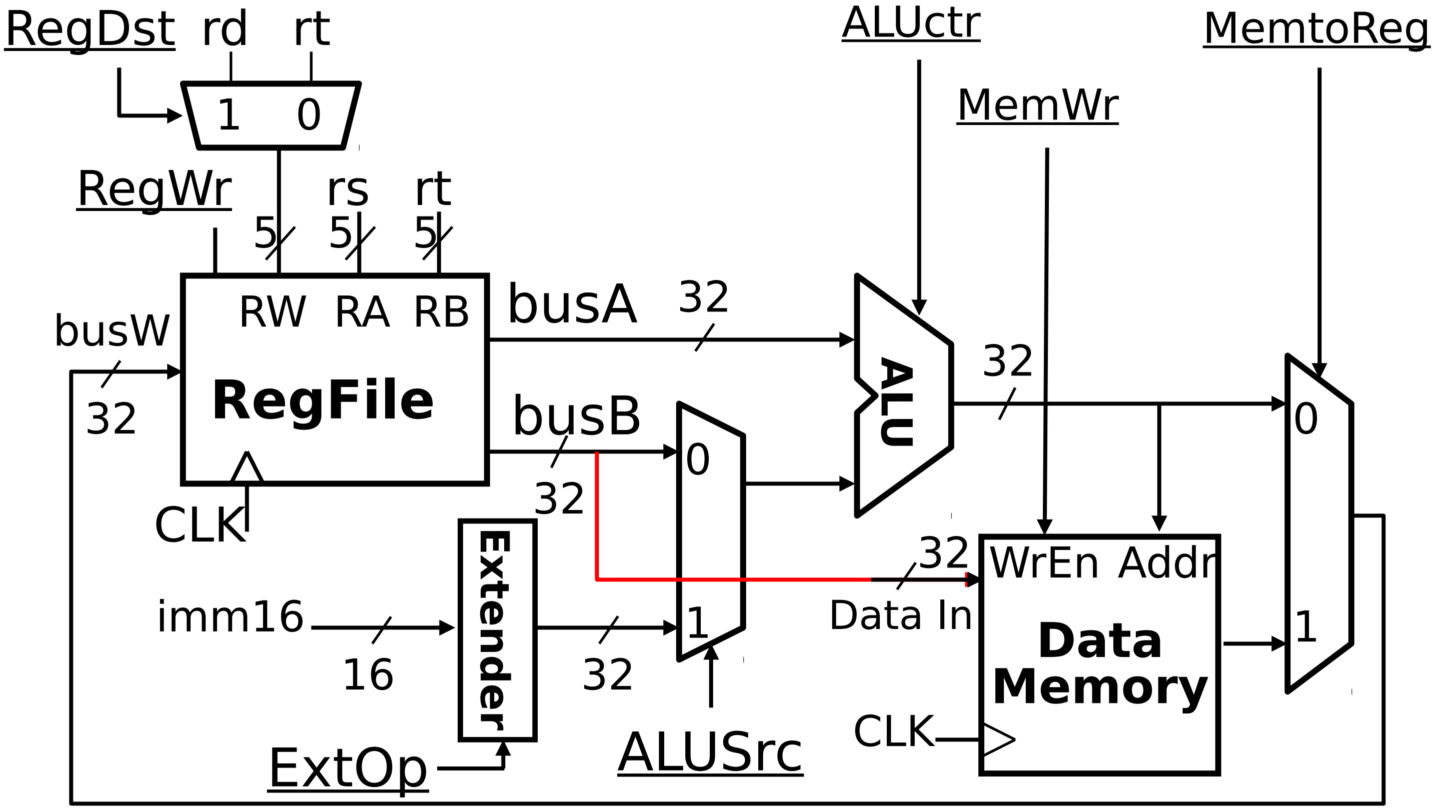


What goes here during a store?

Must now also handle sign extension

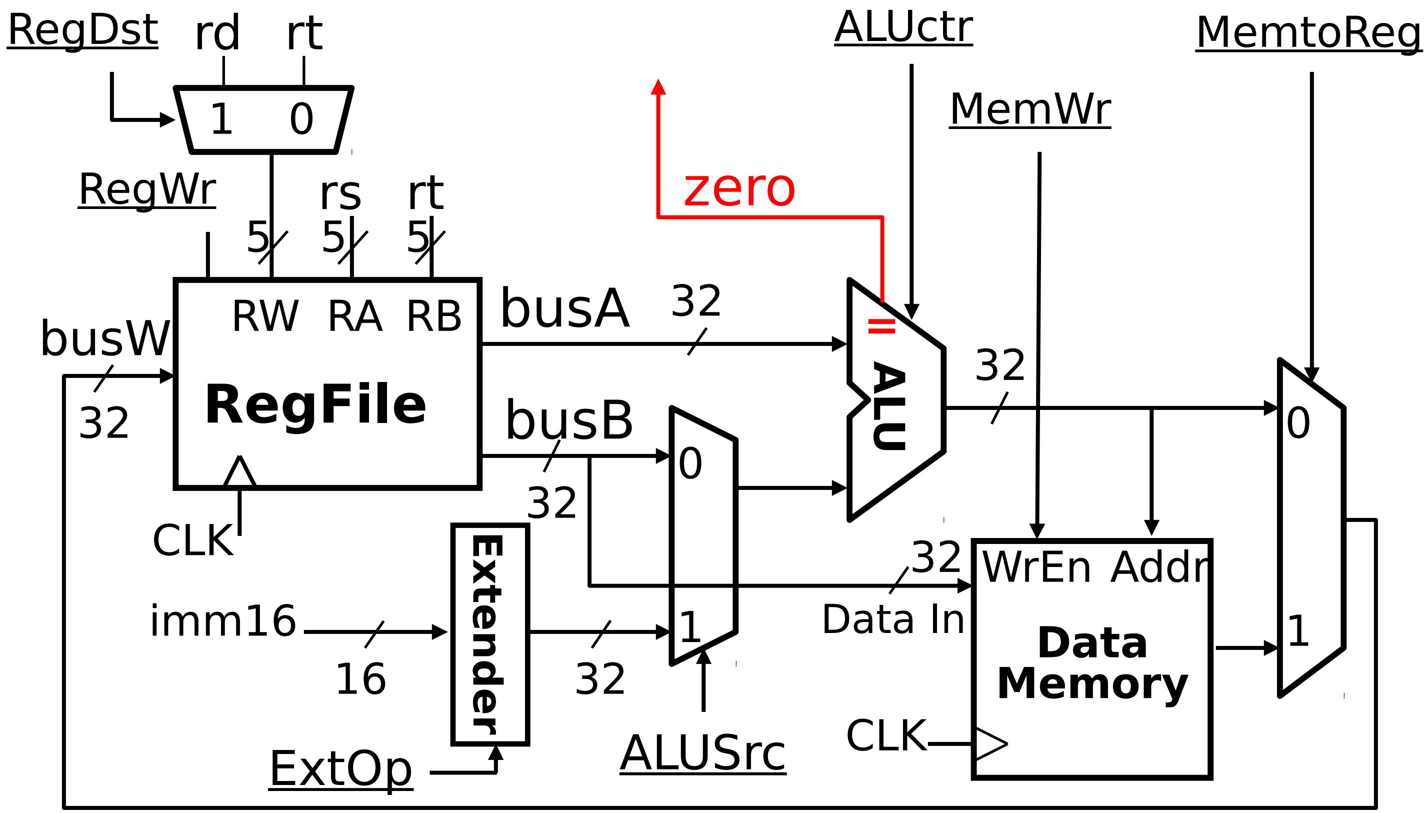
Step 3: Store

- `STORE MEM[R[rs]+sign_ext(Imm16)]←R[rt];`
- Connect `busB` to `Data In` (no extra control needed!)



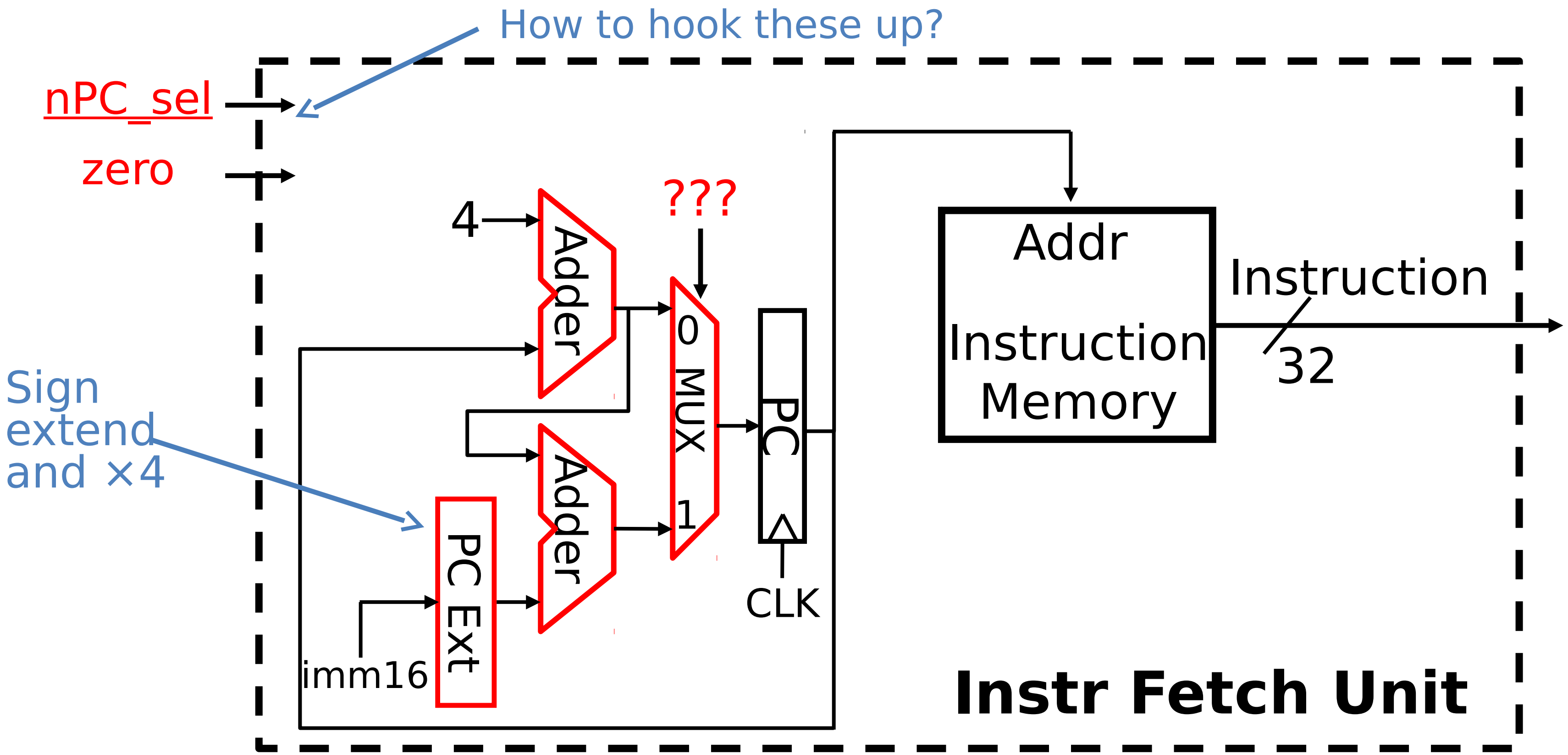
Step 3: Branch If Equal

- BEQ if(R[rs]==R[rt]) then PC←PC+4 + (sign_ext(Imm16) || 00)
- Need comparison output from ALU



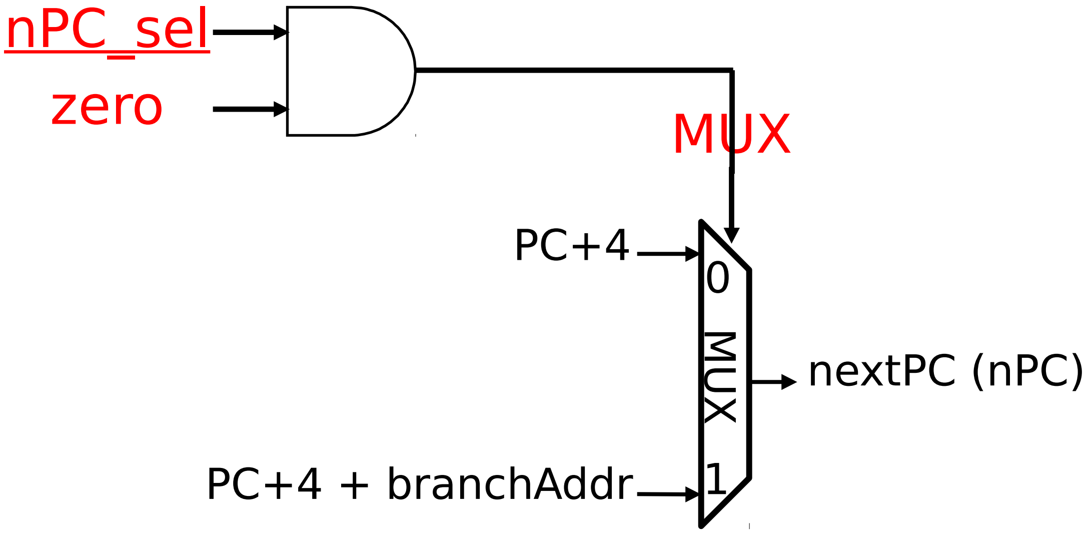
Step 3: Branch If Equal

- BEQ if(R[rs]==R[rt]) then PC←PC+4 + (sign_ext(Imm16) || 00)
- Revisit "next address logic":



Step 3: Branch If Equal

- BEQ if(R[rs]==R[rt]) then PC←PC+4 + (sign_ext(Imm16) || 00)
- Revisit "next address logic":
  - nPC_sel should be 1 if branch, 0 otherwise

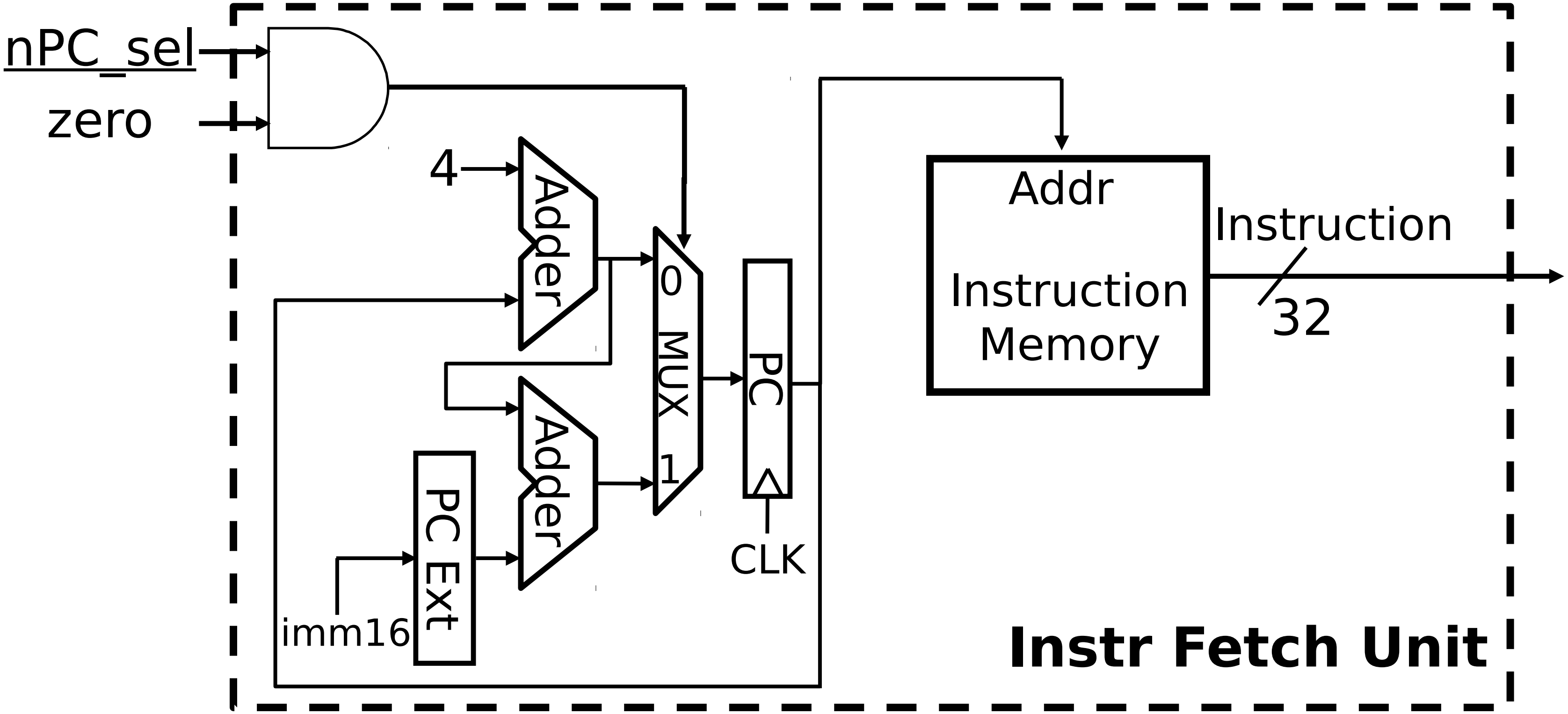


nPC_sel	zero	MUX
0	0	0
0	1	0
1	0	0
1	1	1

How does this change if we add bne?

Step 3: Branch If Equal

- BEQ if(R[rs]==R[rt]) then PC←PC+4 + (sign_ext(Imm16) || 00)
- Revisit "next address logic":



Technology Break

Agenda

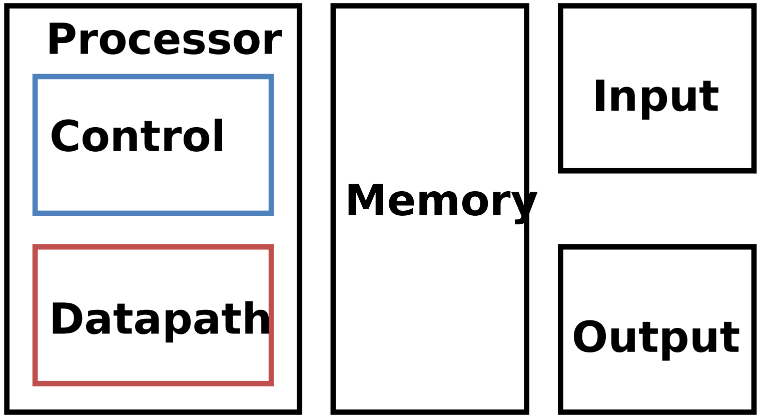
- Processor Design
- Administrivia
- Datapath Overview
- Assembling the Datapath
- Control Introduction

Processor Design Process

- Five steps to design a processor:
  1. Analyze instruction set -> datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements

Now

  4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer
  5. Assemble the control logic
     - Formulate Logic Equations
     - Design Circuits

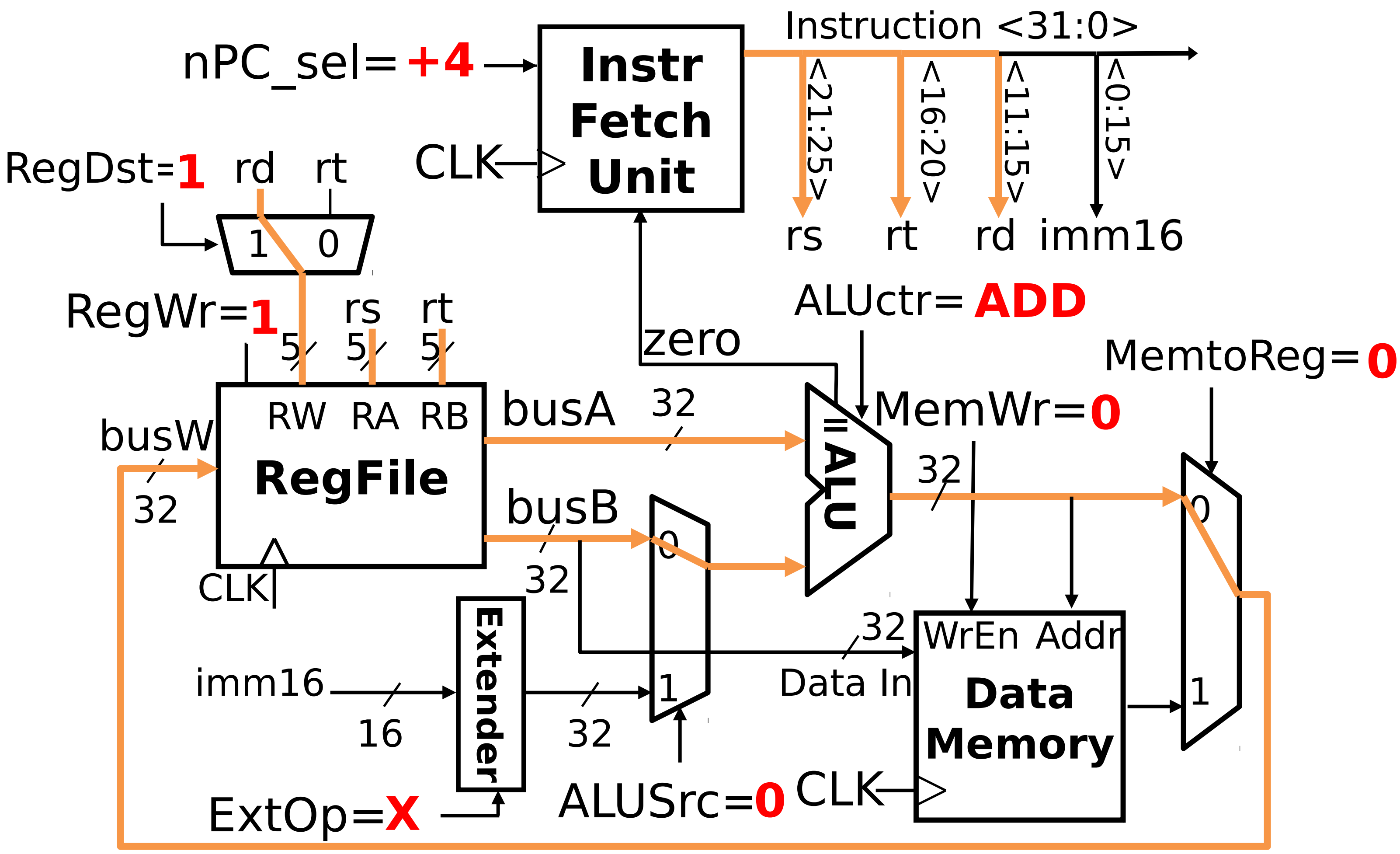

Processor
Control
Datapath
Memory
Input
Output

Control

- Need to make sure that correct parts of the datapath are being used for each instruction
  - Have seen *control signals* in datapath used to select inputs and operations
  - For now, focus on what value each control signal should be for each instruction in the ISA
    - Next lecture, we will see how to implement the proper combinational logic to implement the control

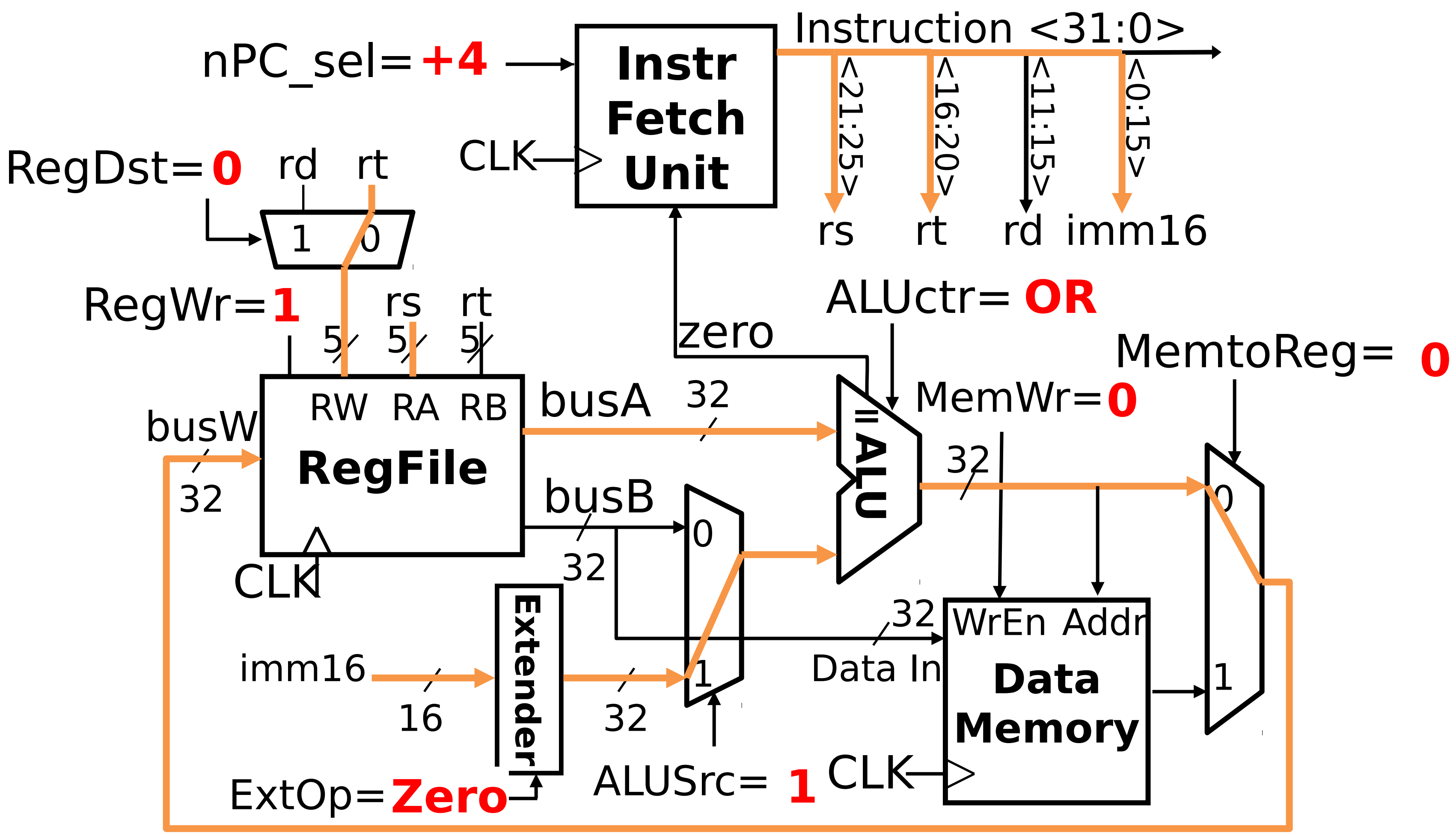
Desired Datapath For addu

- `R[rd]←R[rs]+R[rt];`



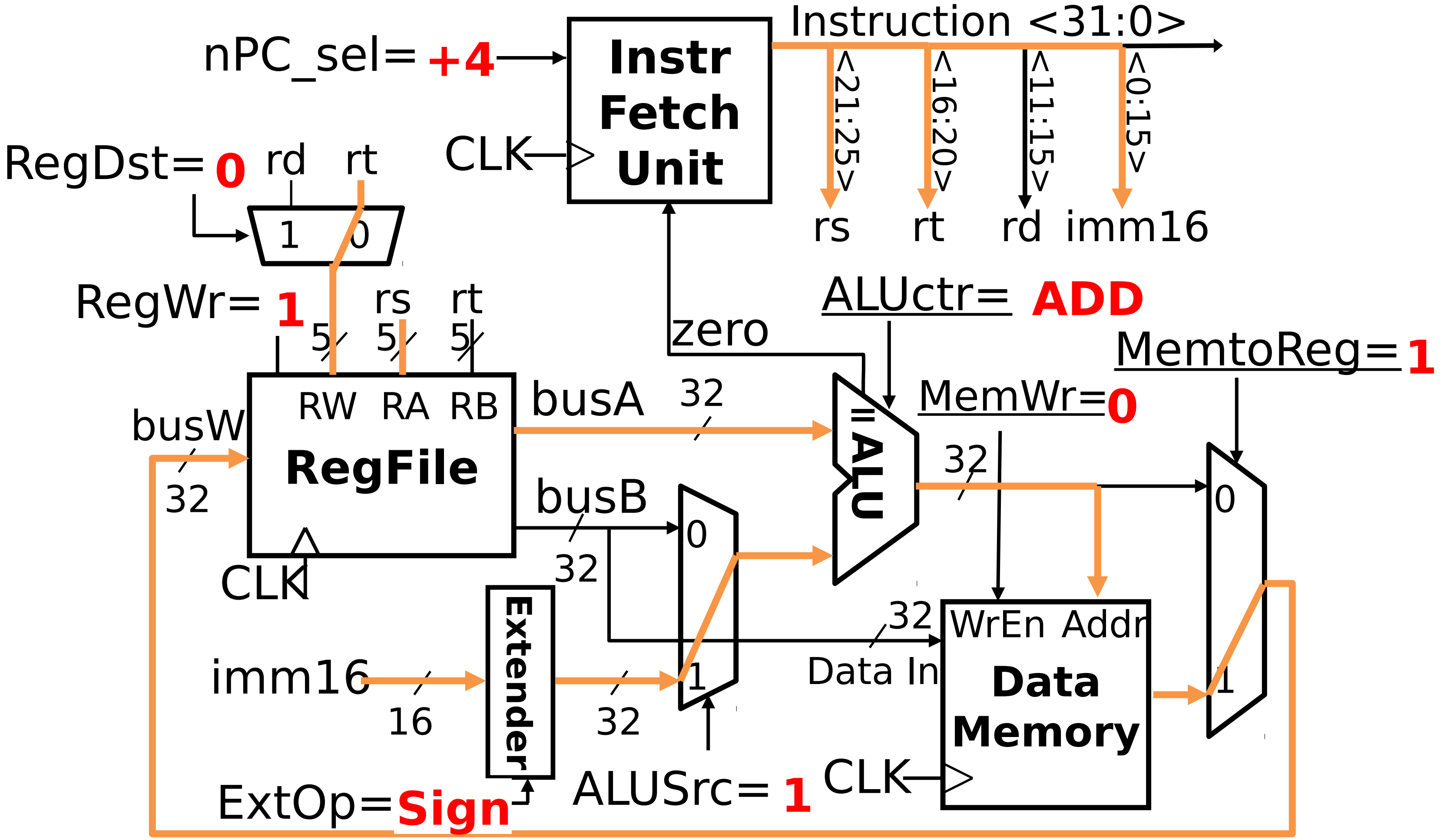
Desired Datapath For ori

- R[rt]←R[rs]|ZeroExt(imm16);



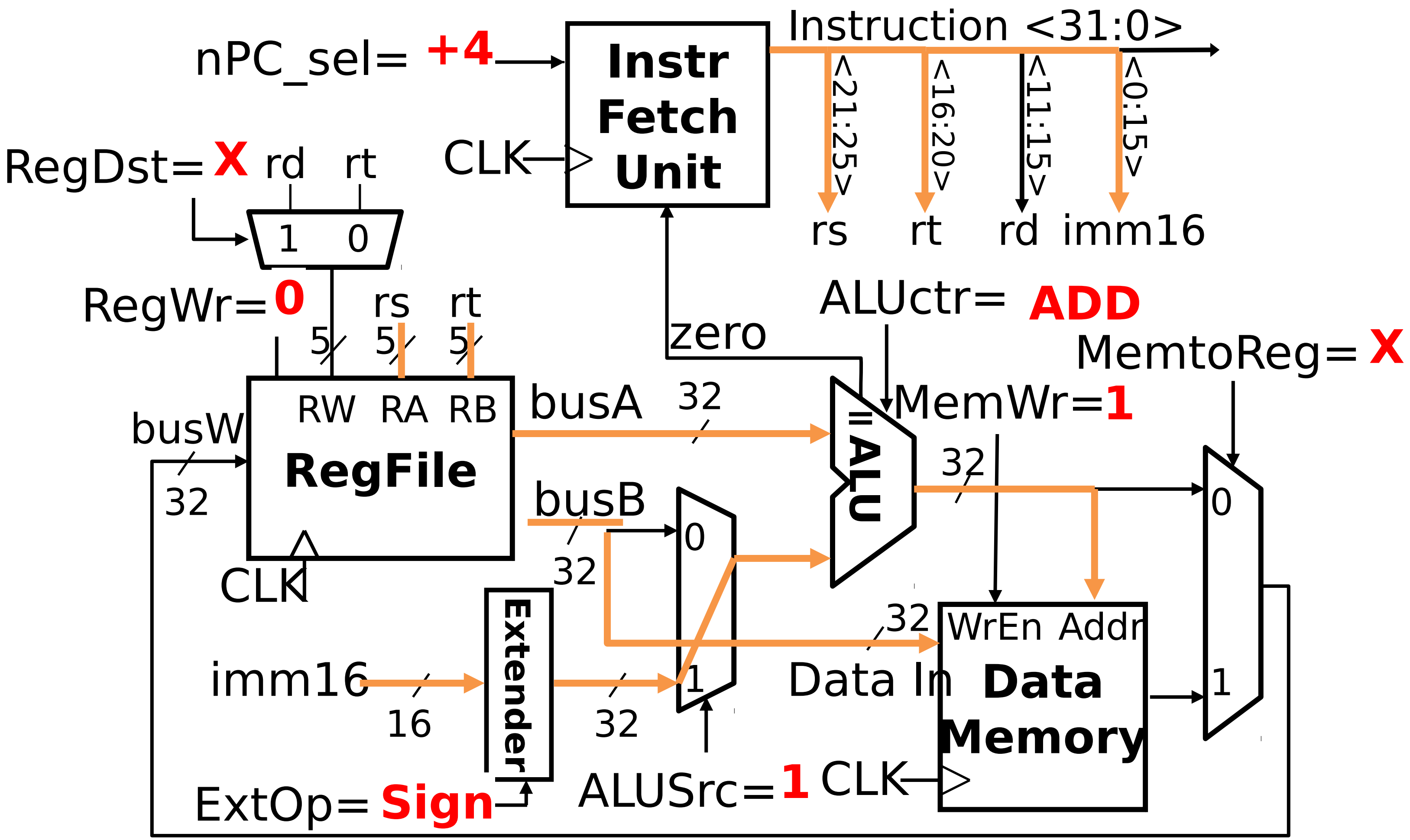
Desired Datapath For load

- R[rt]←MEM{R[rs]+SignExt[imm16]};



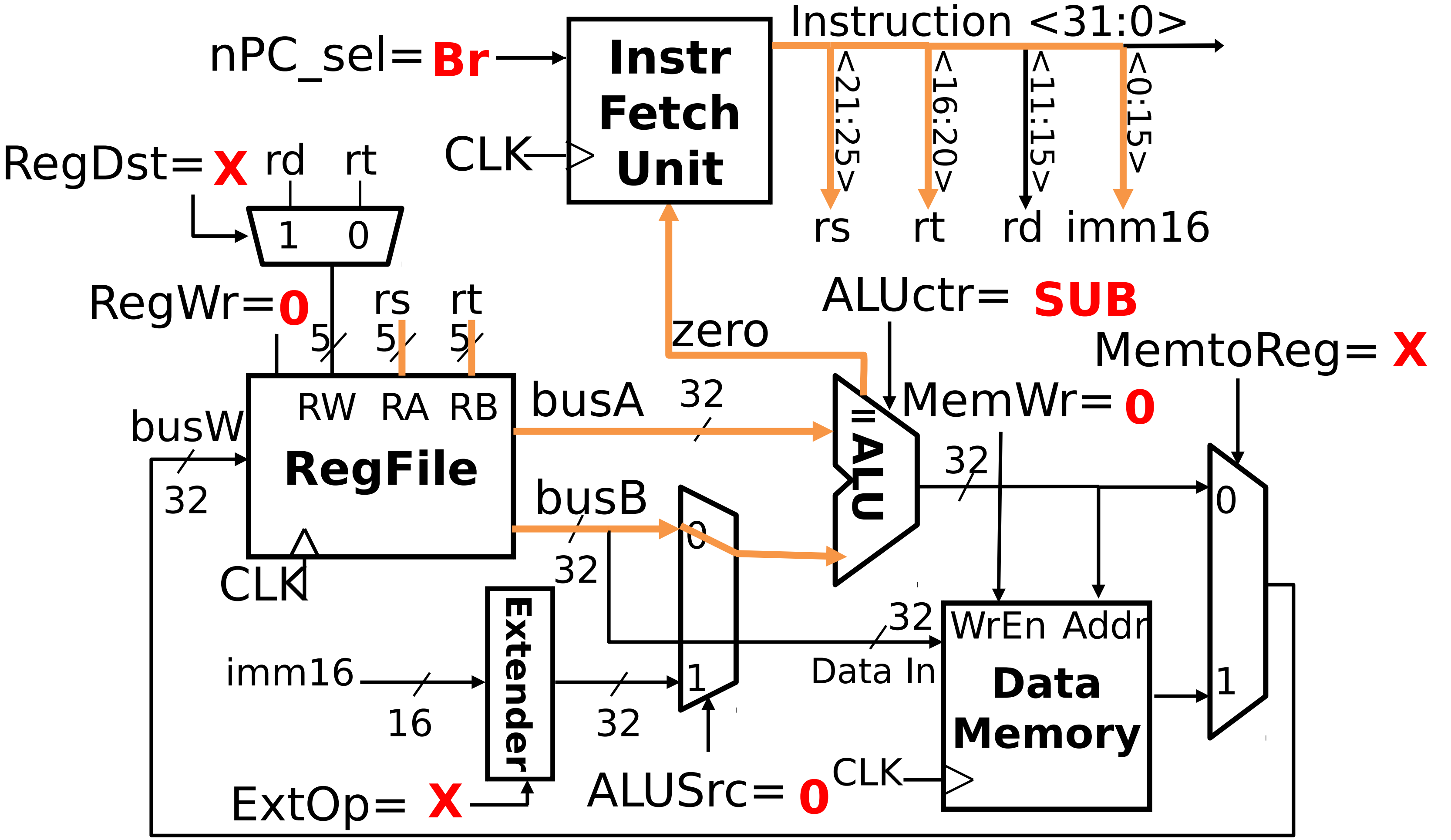
Desired Datapath For store

- MEM{R[rs]+SignExt[imm16]}←R[rt];



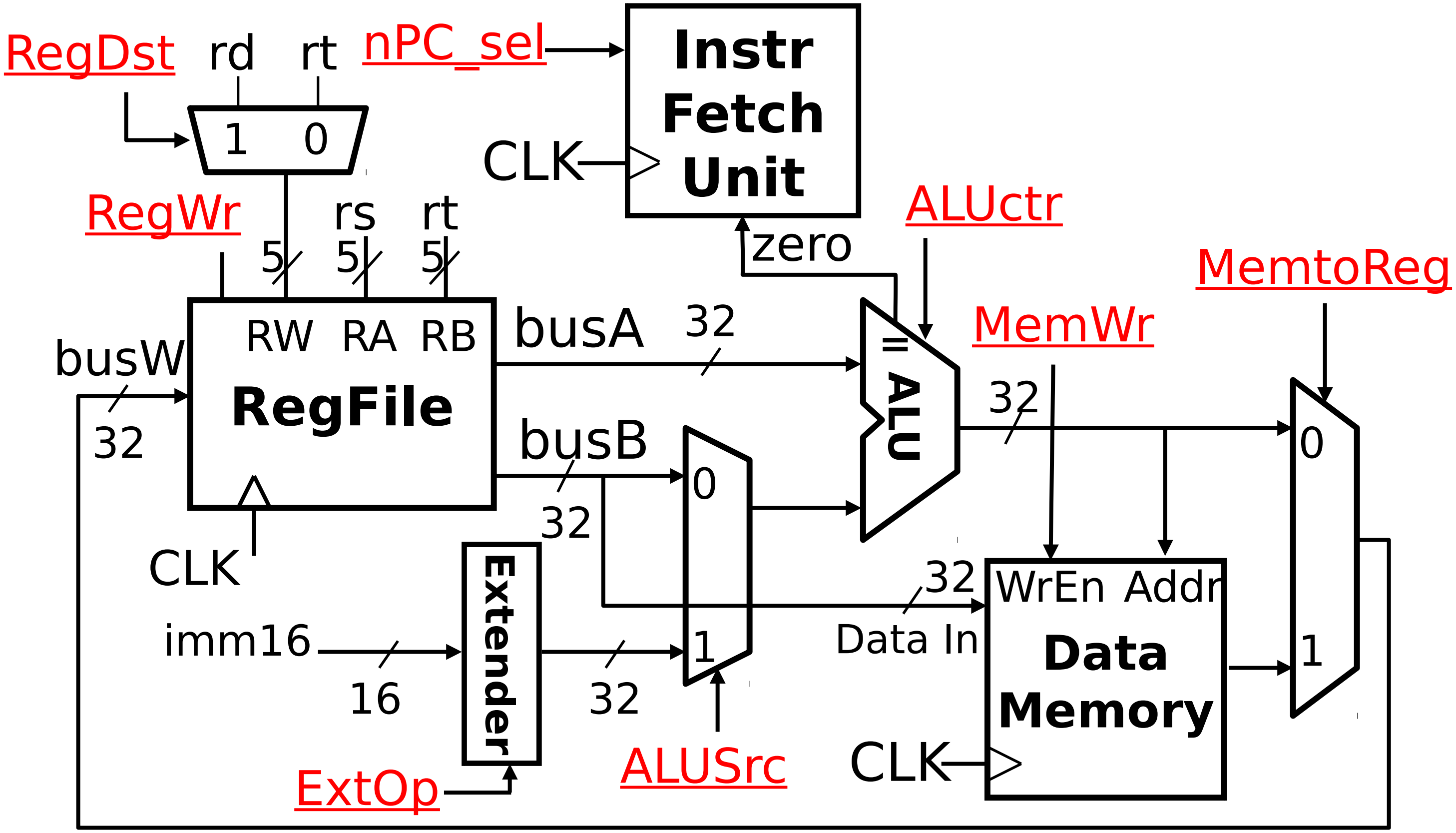
Desired Datapath For beq

- BEQ if(R[rs]==R[rt]) then PC←PC+4 + (sign_ext(Imm16) || 00)



MIPS-lite Datapath Control Signals

ExtOp:	0 **->** “zero”; 1 **->** “sign”	**MemWr:**	1 **->** write memory
ALUsrc:	0 **->** busB; 1 **->** imm16	**MemtoReg:**	0 **->** ALU; 1 **->** Mem
ALUctr:	“ADD”, “SUB”, “OR”	**RegDst:**	0 **->** “rt”; 1 **->** “rd”
nPC_sel:	0 **->** +4; 1 **->** branch	**RegWr:**	1 **->** write register



Question: Which statement is TRUE about the MIPS-lite ISA?

(B) When not in use, parts of the datapath cease to carry a value.

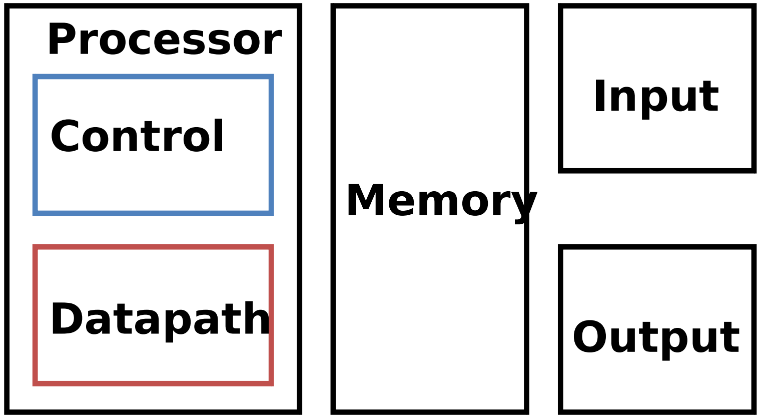
(G) Adding the instruction `jr` will not change the datapath.

(P) All control signals will be don’t care (‘X’) in at least one instruction.

(Y) **Adding the instruction `addiu` will not change the datapath.**

Summary (1/2)

- Five steps to design a processor:
  - Analyze instruction set -> datapath requirements
  - Select set of datapath components & establish clock methodology
  - Assemble datapath meeting the requirements
  - Analyze implementation of each instruction to determine setting of control points that effects the register transfer
  - Assemble the control logic
    - Formulate Logic Equations
    - Design Circuits



Summary (2/2)

- Determining control signals
  - Any time a datapath element has an input that changes behavior, it requires a control signal
    (e.g. ALU operation, read/write)
  - Any time you need to pass a different input based on the instruction, add a MUX with a control signal as the selector
    (e.g. next PC, ALU input, register to write to)
- Your datapath and control signals will change based on your ISA