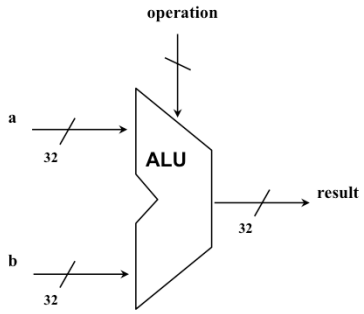
Let’s build a processor

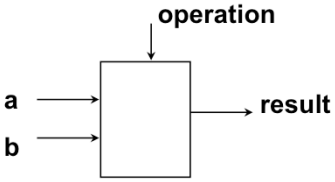


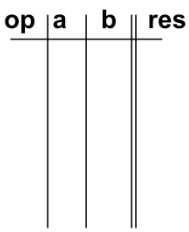
Other handouts

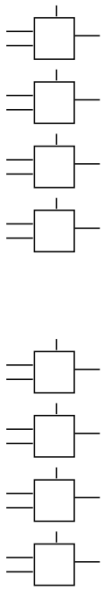
To handout next time

An ALU (arithmetic logic unit)

- Let's build an ALU to support the `andi` and `ori` instructions
  - we'll just build a 1 bit ALU, and use 32 of them





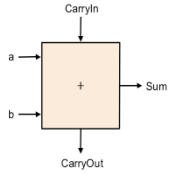


- Possible Implementation (sum-of-products):
  - Programmable logic array (PLA)

$$D = (\overline{A} \cdot \overline{B} \cdot C) + (\overline{A} \cdot B \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot B \cdot C)$$

Different Implementations

- Not easy to decide the "best" way to build something
  - Don't want too many inputs to a single gate
  - Don't want to have to go through too many gates
  - For our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:



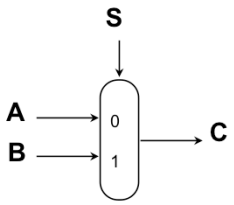
$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$
$sum = a \text{ xor } b \text{ xor } c_{in}$

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

- How could we build a 1-bit ALU for **add**, **and**, and **or**?
- How could we build a 32-bit ALU?

Review: The Multiplexor

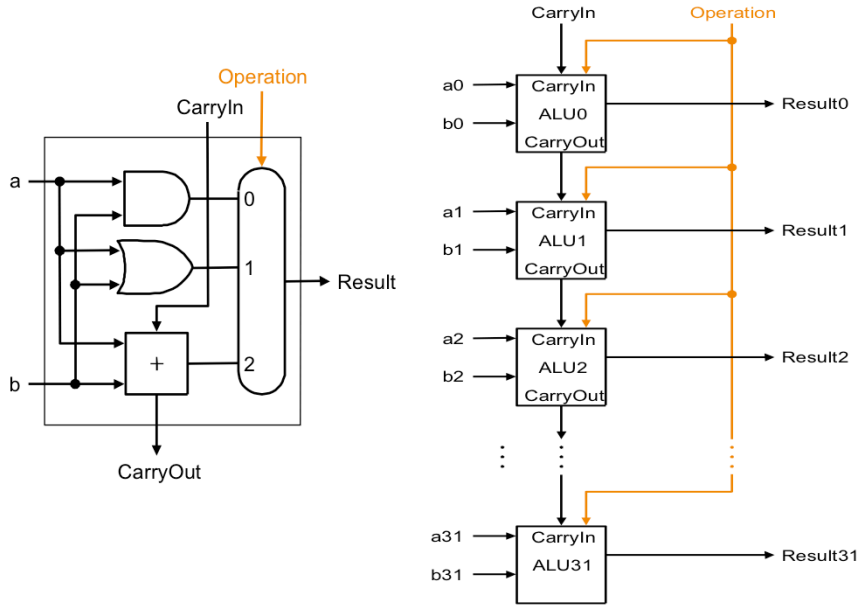
- Selects one of the inputs to be the output, based on a control input



note: we call this a 2-input mux even though it has 3 inputs!

- Let's build our ALU using a MUX:

Building a 32 bit ALU



A 1 bit ALU that performs AND OR and addition

Configuration called ripple carry

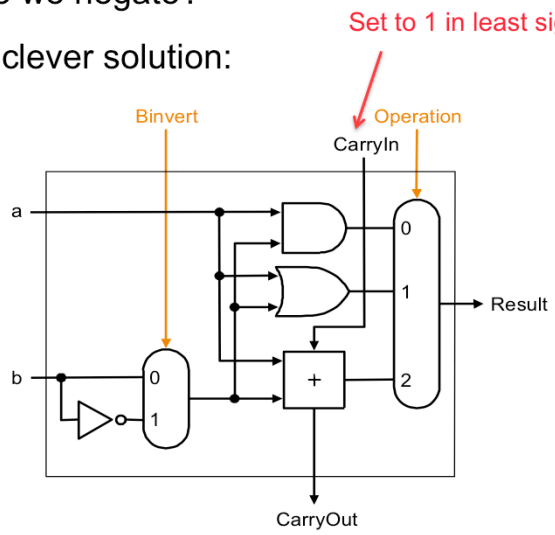
It is not efficient because each unit must wait for the one above to complete the calculation (requires synch)

Other schemes are the carry-lookahead adder where groups of bits try to predict with additional logic whether

a carry will be generated in the previous block or not.

What about subtraction (a – b) ?

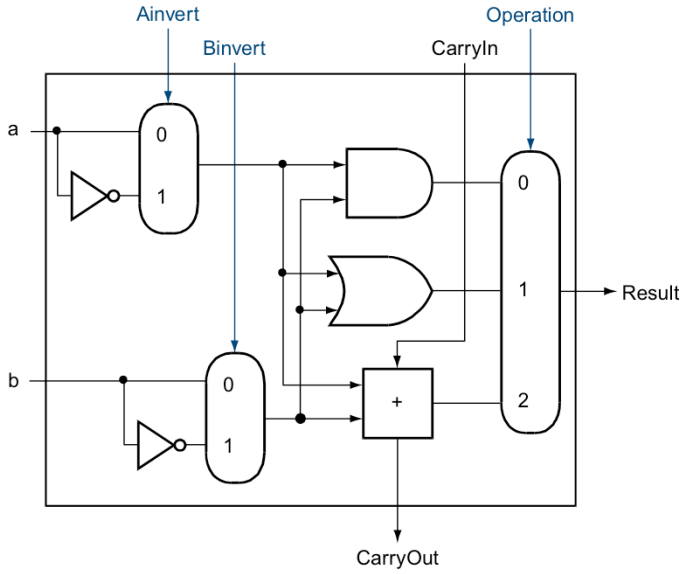
- Two's complement approach: just negate b and add to a.
- How do we negate?
- A very clever solution:



One‘s complement is the inversion of all the bits.
Two‘s complement is the one‘s complement + 1.

Adding a NOR function

- Can also choose to invert a. How do we get “a NOR b” ?



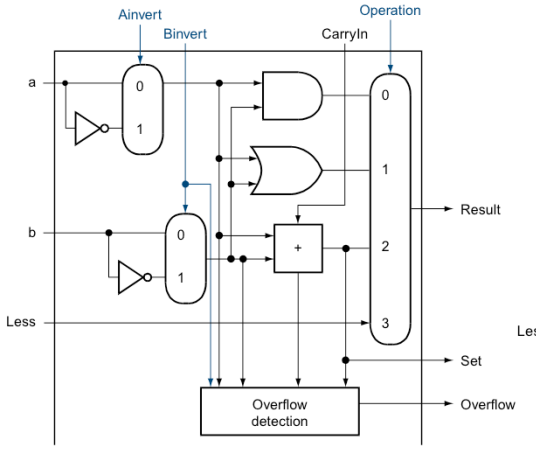
NOR = NOT A AND NOT B

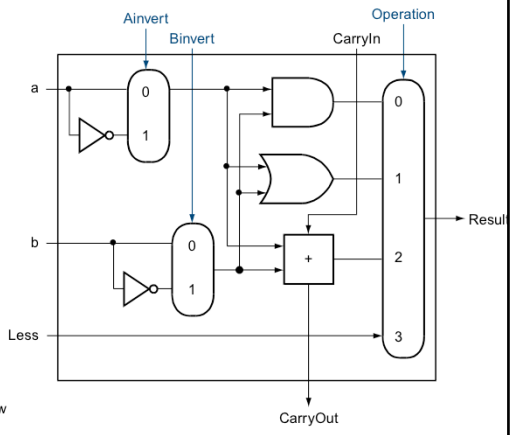
Tailoring the ALU to the MIPS

- Need to support the set-on-less-than instruction (slt)
  - remember: slt is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - use subtraction: (a-b) < 0 implies a < b
- Need to support test for equality (beq $t5, $t6, Lbl)
  - use subtraction: (a-b) = 0 implies a = b

Supporting slt

- Can we figure out the idea?





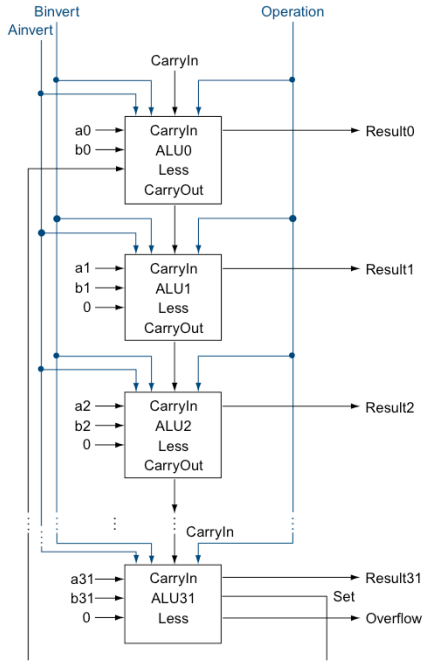
Use this ALU for most significant bit

all other bits

LESS sets all bits except for least significant one to 0

Sign is the SLT bit (sign output of adder)

Supporting slt

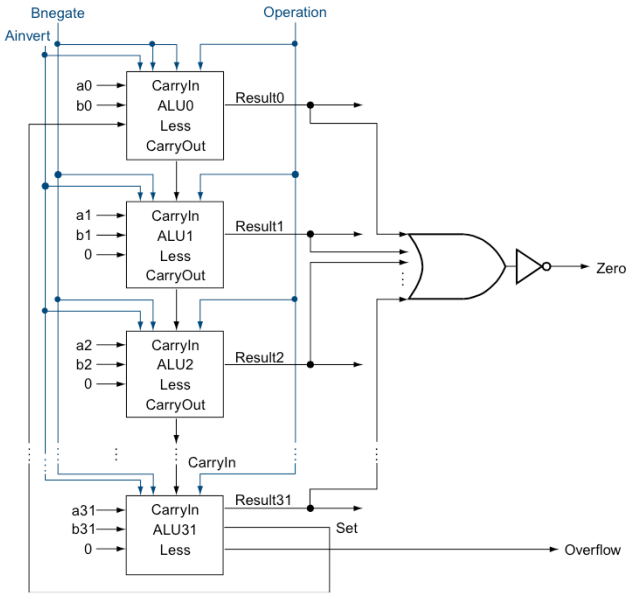


Test for equality

- Notice control lines:

```
0000 = and
0001 = or
0010 = add
0110 = subtract
0111 = slt
1100 = NOR
```

Note: Zero is a 1
when the result is zero!



To add branch on equal (or not equal) need extra logic in the output
BE is implemented by OR and then NOT

Conclusion

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU

- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series (on the "critical path" or the "deepest level of logic")

- Our primary focus: comprehension, however:
  - Clever changes to organization can improve performance (similar to using better algorithms in software)

ALU Summary

- We can build an ALU to support MIPS addition
- Our focus is on comprehension, not performance
- Real processors use more sophisticated techniques for arithmetic (e.g. do not use ripple carry adder)
- Where performance is not critical, hardware description languages allow designers to completely automate the creation of hardware!

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes anywhere
   always @(ALUctl, A, B) //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1:0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0; //default to 0, should not happen;
      endcase
endmodule
```

FIGURE B.4.3 A Verilog behavioral definition of a MIPS ALU. This could be synthesized using a module library containing basic arithmetic and logical operations.

In the ripple carry adder the MSB must wait for all others to be computed; this is not efficient

Other designs are possible

Basic MIPS Architecture Review

[Adapted from Mary Jane Irwin for
Computer Organization and Design,
Patterson & Hennessy, © 2005, UCB]

Other handouts
To handout next time

We look at MIPS architecture i.e. methods and techniques to Implement a processor. First we introduce a simple architecture and then a more realistic one.

Review: The Performance Equation

- Our basic performance equation is then

$$\text{CPU time} = \text{Instruction\_count} \times \text{CPI} \times \text{clock\_cycle}$$

or

$$\text{CPU time} = \frac{\text{Instruction\_count} \times \text{CPI}}{\text{clock\_rate}}$$

- These equations separate the three key factors that affect performance
  - Can measure the CPU execution time by running the program
  - The clock rate is usually given in the documentation
  - Can measure instruction count by using profilers/simulators without knowing all of the implementation details
  - CPI varies by instruction type and ISA implementation for which we must know the implementation details

CPI = Clock Cycles Per Instruction (the average number of clock cycles each instruction takes to execute)

Note that instruction count is dynamic – i.e., its not the number of lines in the code, but THE NUMBER OF INSTRUCTIONS EXECUTED

The implementation of the processor determines both the clock cycle time and the number of clock cycles per instruction.

So the first area of craftsmanship is in trading function for size. … The second area of craftsmanship is space-time trade-offs. For a given function, the more space, the faster.

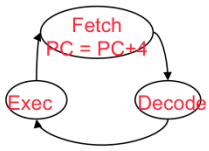
The Mythical Man-Month, Brooks, pg. 101

For a fixed time, more function requires more space

For a fixed function, execution can be faster if more space is devoted

The Processor: Datapath & Control

- Our implementation of the MIPS is simplified
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- Generic implementation
  - use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
  - decode the instruction (and read registers)
  - execute the instruction



- All instructions (except `j`) use the ALU after reading the registers

How? memory-reference? arithmetic? control flow?

Core implementation (so we consider only a few important instructions)
For each instruction two steps are required: 1) fetch instruction via PC; 2) Read 1 or 2 registers
After these two steps the actions depend on the instruction class.

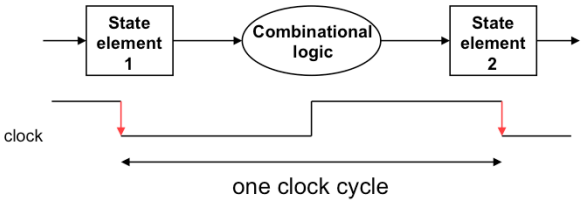
memory reference uses ALU to compute addresses

arithmetic uses the ALU to do the require arithmetic

control uses the ALU to compute branch conditions.

Clocking Methodologies

- The clocking methodology defines when signals can be read and when they are written
  - An edge-triggered methodology
- Typical execution
  - read contents of state elements
  - send values through combinational logic
  - write results to one or more state elements



- Assumes state elements are written on every clock cycle; if not, need explicit write control signal
  - write occurs only when both the write control is asserted and the clock edge occurs

There are several components in a design: combinational (AND OR ALU etc gives the same output given the same inputs (no memory)) and

state elements (has internal storage eg instruction and data memories, registers etc).

State elems have at least two inputs and one output (input: data to be written and clock which says when to write)

The output provides the value that was written in an earlier clock cycle.

Logic components with a state are called sequential because their outputs depend on inputs and state

A state element can be read anytime.

Asserted == high

Hence, clocking needed to avoid writing and reading at the same time

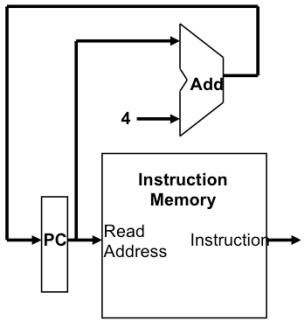
Edge-triggered means that updates happen only when the clock signal transitions from hi to lo or vice versa

Because only state elements can store data, a combinational logic must take input from state el and output to state el

Clock cycle must be long enough to have a stable output at the

Fetching Instructions

- Fetching instructions involves
  - reading the instruction from the Instruction Memory
  - updating the PC to hold the address of the next instruction



  - PC is updated every cycle, so it does not need an explicit write control signal
  - Instruction Memory is read every cycle, so it doesn't need an explicit read control signal

First elements of a datapath (memory, PC register and added)

Add is an ALU fixed as adder

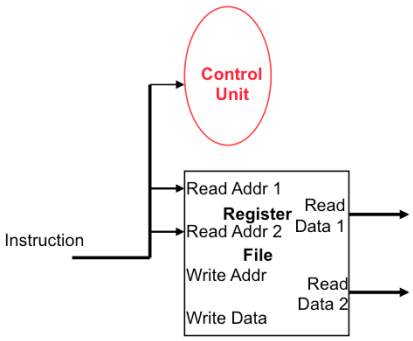
Instruction memory does not need write bus because the datapath does not write instructions

Because of this the memory is considered as combinatorial logic for now (at some stage a program must

be loaded)

Decoding Instructions

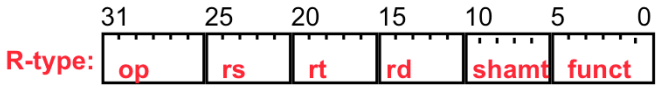
- Decoding instructions involves
  - sending the fetched instruction's opcode and function field bits to the control unit



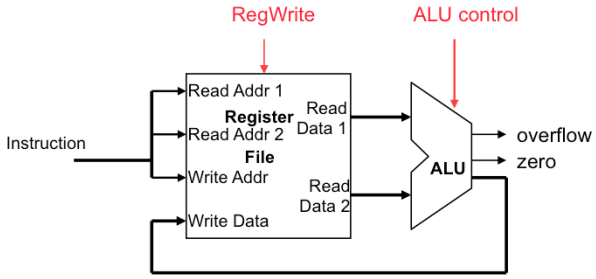
  - reading two values from the Register File
    - Register File addresses are contained in the instruction

Executing R Format Operations

- R format operations (`add, sub, slt, and, or`)



- perform the (op and funct) operation on values in rs and rt
- store the result back into the Register File (into location rd)



- The Register File is not written every cycle (e.g. `sw`), so we need an explicit write control signal for the Register File

Color lines used to show control signals

Recall that R-type instruct have 3 operands, two inputs and one output

The read outputs always show the content of the registers specified in the two inputs.

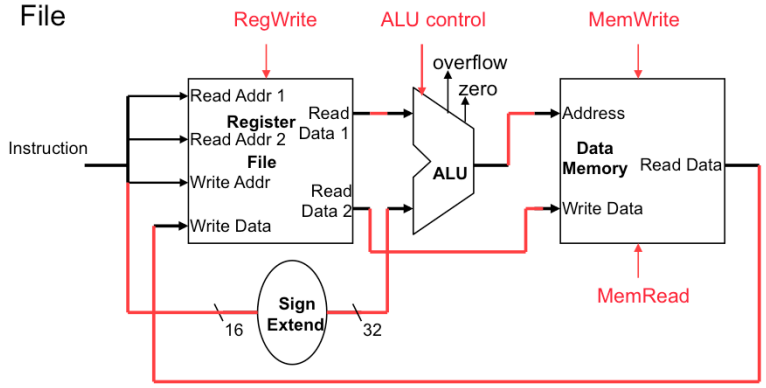
Writes instead are controlled by the write control signal (regwrite)

We need write address and write data

Remember that the logic is such that read and write can happen within the same cycle. However, the basic structure is that read provides the current value to the output and written values will be available only from the next cycle on

Executing Load and Store Operations

- Load and store operations involves
  - compute memory address by adding the base register (read from the Register File during decode) to the 16-bit signed-extended offset field in the instruction
  - store value (read from the Register File during decode) written to the Data Memory
  - load value, read from the Data Memory, written to the Register File



We need to add data memory and sign extend for load and write

Data memory needs both read and write control signals

Only one of read and write can be asserted within a clock cycle

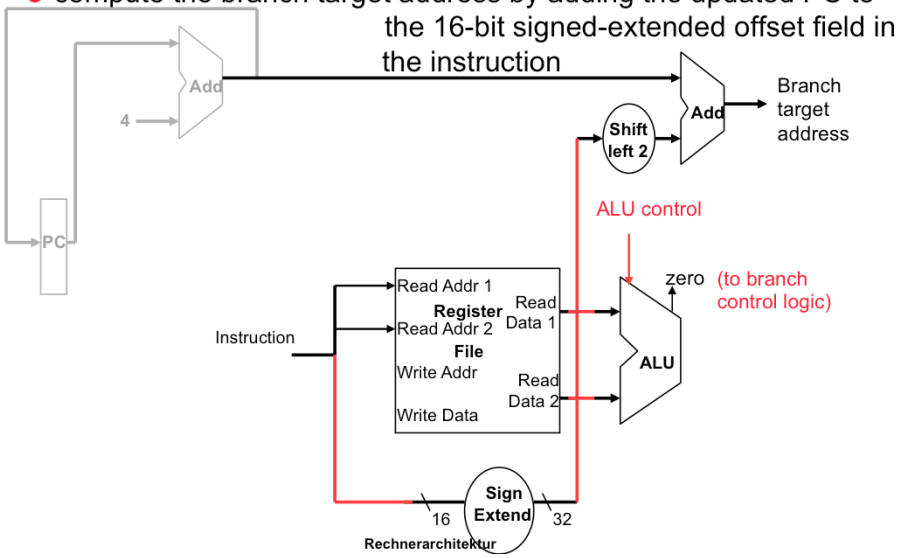
MemRead ist nötig, das es ein Cache Speicher ist und die Address ev gar nicht im Cache ist.

Nur wenn zusätzlich MemRead gestzt ist, muss der Cache entsprechend aktualisiert werden.

PH, p242, Abb 5.8

Executing Branch Operations

- Branch operations involves
  - compare the operands read from the Register File during decode for equality (`zero` ALU output)
  - compute the branch target address by adding the updated PC to the 16-bit signed-extended offset field in the instruction



The base address is the address of the instruction following the branch (i.e., PC+4 which we can take from the FETCH datapath)

The offset field is shifted left 2 bits so that it is a word offset; this shift increases the effective range of the offset field by a factor of 4

Shifting is simply a re-routing of the input output wires plus an addition of two 00 wires

No shift hardware is needed since the shift is fixed

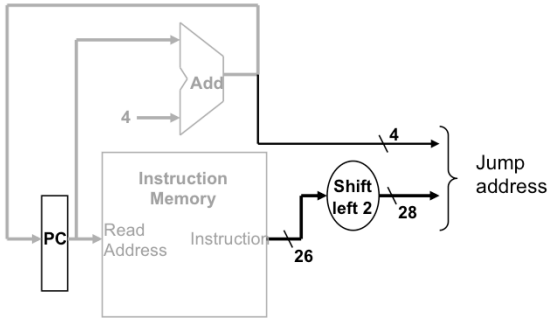
Since the offset was 16 bits we only throw away sign bits

When condition is true we say that the BRANCH IS TAKEN

Describe branch logic: for example beq $t1, $t1, offset

Executing Jump Operations

- Jump operation involves
  - replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits



Creating a Single Datapath from the Parts

- Assemble the datapath segments and add control lines and multiplexors as needed
- Single cycle design – fetch, decode and execute each instruction in one clock cycle
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)
  - multiplexors needed at the input of shared elements with control lines to do the selection
  - write signals to control writing to the Register File and Data Memory
- Cycle time is determined by length of the longest path

We have examined the datapaths of the individual instruction classes

Now we can combine then into a single datapath and add control to complete the implementation

We aim to do this in a single clock cycle

This means that no datapath resources can be used

more than once per instruction

Any element needed more than once must be duplicated

To let two instruction classes share the same datapath element we need multiple connections through a multiplexor before

going to the input and also a control signal to choose input

Example: Building a datapath

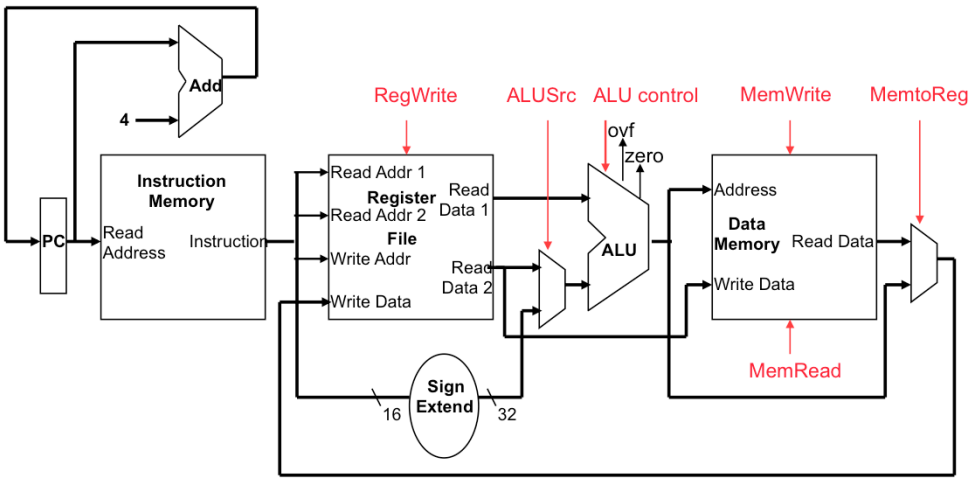
- Arithmetical-logical R-type instructions and memory instructions datapaths are similar
- Differences:
  - R-type use ALU with inputs from registers; memory instructions use ALU as well, but the second input is the offset
  - The value stored in a destination comes from ALU (R-type) or memory (for a load)
- Build a datapath for these two instructions with a single register file and a single ALU (can add multiplexors)

Solution: we must support two different sources for the second ALU input as well as two different sources for the data stored (write) in the register file.

Thus one multiplexor at the second ALU input and one multiplexor at the data input to the register file.

See next slide

Fetch, R, and Memory Access Portions

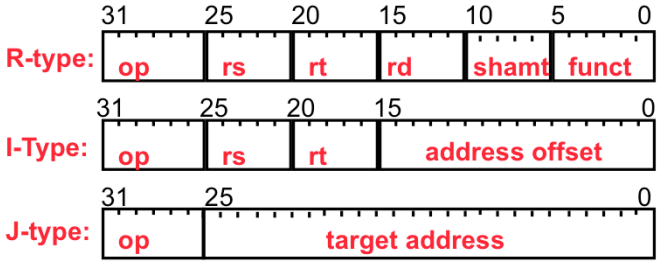


Drawing of all datapath elements together.

Now we can add the control unit.

Adding the Control

- Selecting the operations to perform (ALU, Register File and Memory read/write)
- Controlling the flow of data (multiplexor inputs)



- Observations
  - op field always in bits 31-26
  - addr of registers to be read are always specified by the rs field (bits 25-21) and rt field (bits 20-16); for lw and sw rs is the base register
  - addr. of register to be written is in one of two places – in rt (bits 20-16) for lw; in rd (bits 15-11) for R-type instructions
  - offset for beq, lw, and sw always in bits 15-0

Let us review the 3 types of instructions

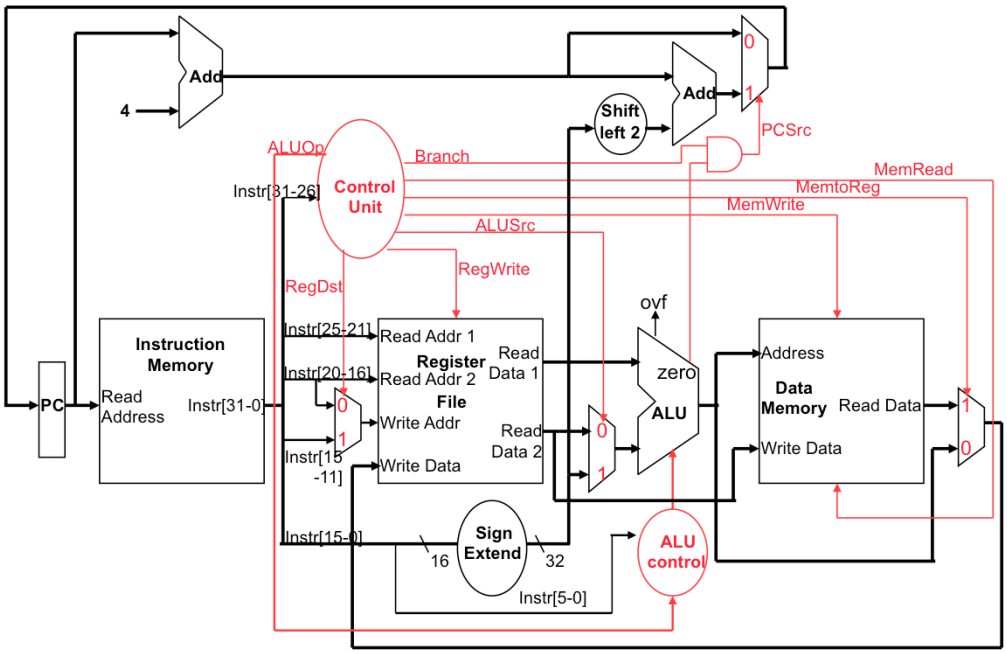
Describe bits and notation for each field type

Destination register (RT) can be either in 20:16 (load) or 15-11 (R-type)
Hence we need a multiplexor to choose

BRANCH is as I-type format
JUMP uses another format

We will use these labels in the next diagram

Single Cycle Datapath with Control Unit



Note additional multiplexor for the PC+4 vs Branching choice

All multiplexors have 2 inputs and hence a single control line is needed

Note mux control inputs have been swapped (for three of the muxes) from the last picture to be consistent with the book.

The control unit must be able to take inputs and generate the write signals for each state element,

The selector control for each multiplexor, and the ALU control

The ALU control:

Recall that ALU defines AND OR ADD SUB SLT NOR

For load or store word we use ALU to compute address via ADD

For R-type instructions the ALU needs one of the five: AND OR SUB ADD SLT depending on the funct part of the instruction

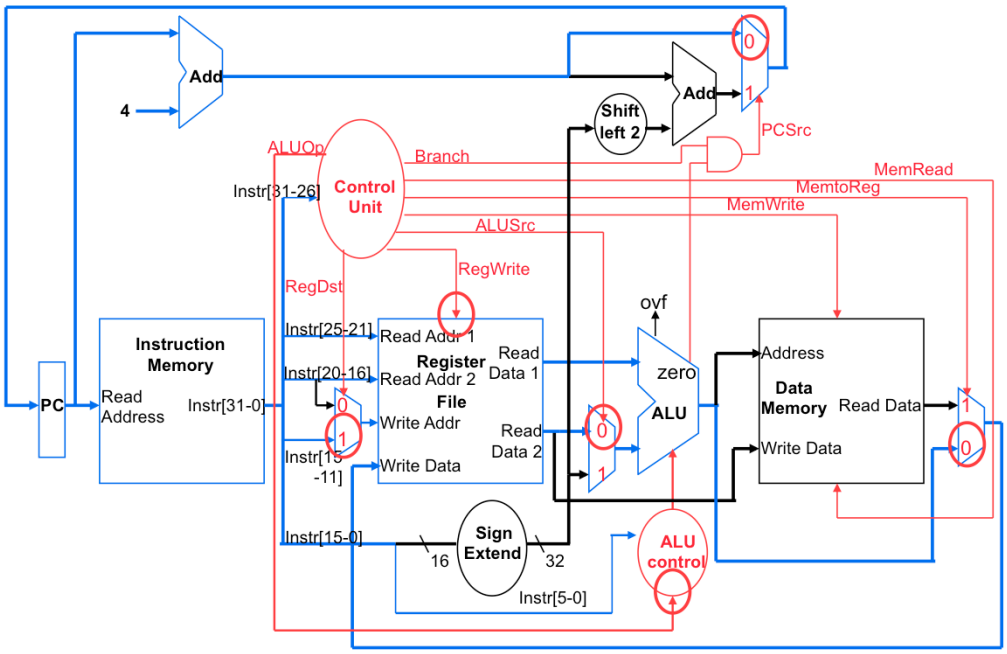
For branch on equal ALU needs a SUB

ALU gets as input 5 funct bits + ALUOp

ALUOp is the ALU control input (2 bits) 00 – ADD, 01 – SUB, 10 – funct (determined by).

Output of ALU control unit is 4 bit

R-type Instruction Data/Control Flow

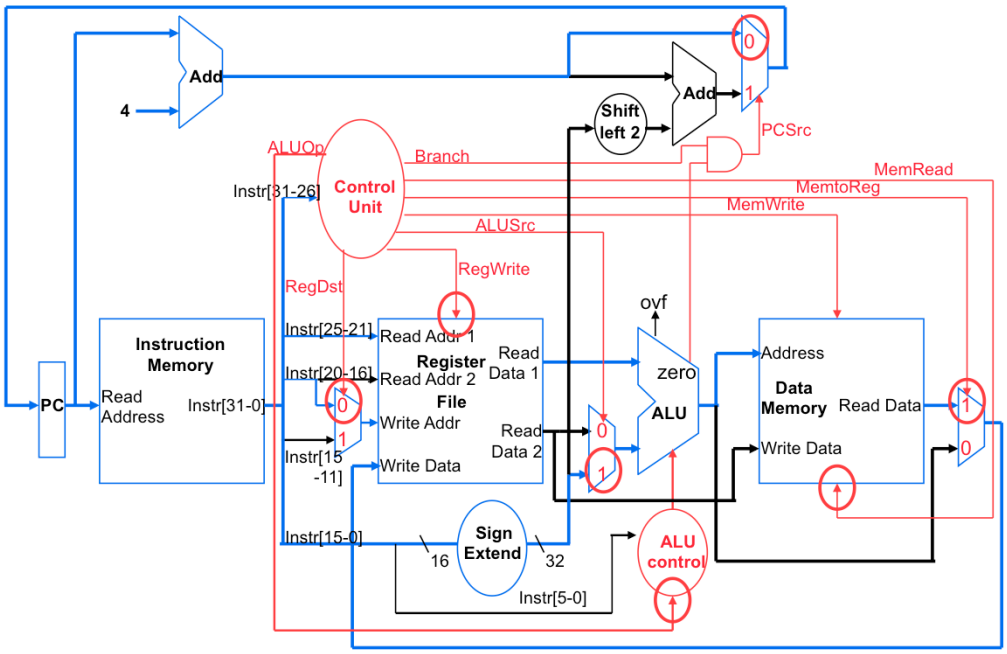


For example ADD $t1, $t2, $t3

Four steps to execute instruction

1. Instr is fetched and PC incremented
2. T2 and t3 are read from register while the MAIN CONTROL UNIT prepares the settings for the control lines
3. ALU performs calculation (ADD) using the function code funct
4. Result from ALU is written into register t1

Load Word Instruction Data/Control Flow

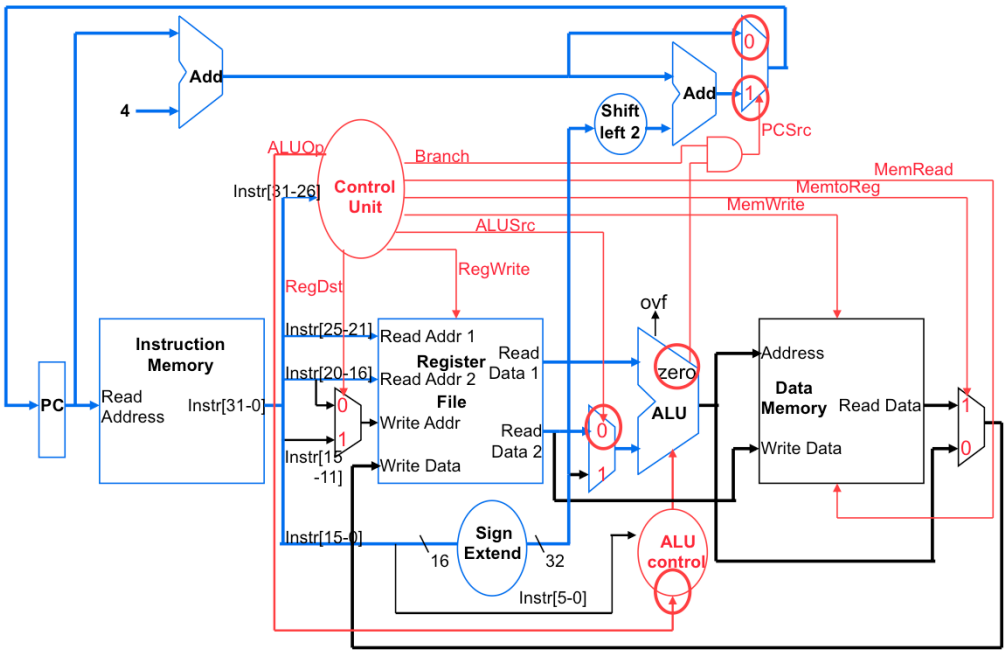


LW $t1, offset($t2)

Five steps:

1. Fetch and PC+4
2. T2 read from register file
3. ALU computes sum of register t2 + offset
4. Sum is used as address for the data memory
5. Data from memory is written to register t1

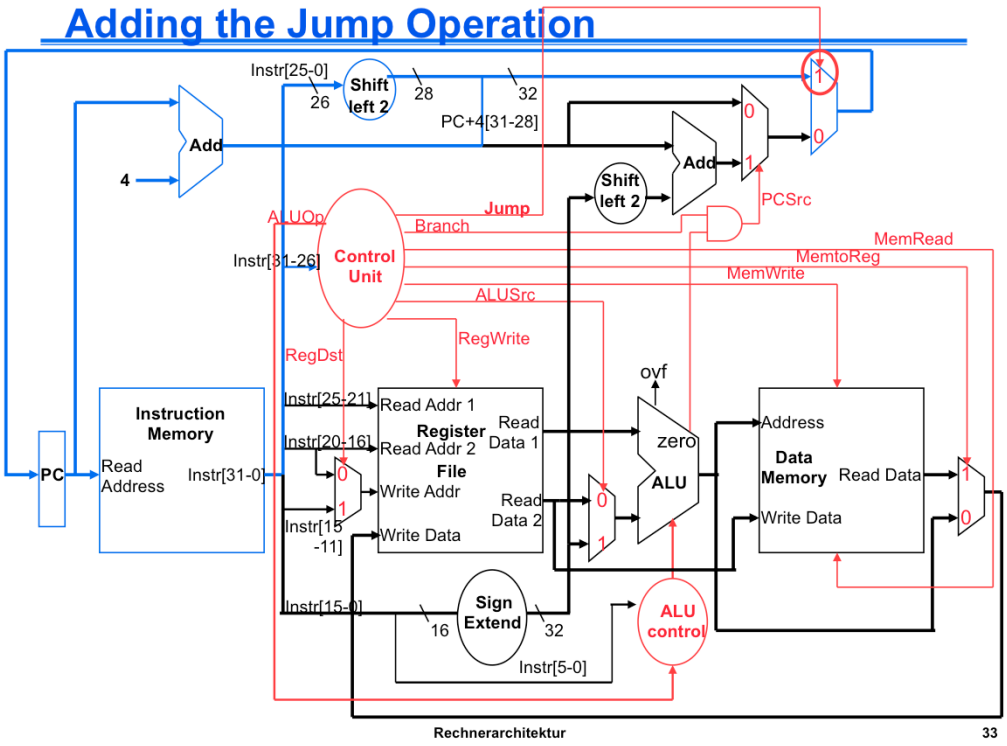
Branch Instruction Data/Control Flow



BEQ t1, t2, offset

Four steps

1. Fetch PC+4
2. T1 and T2 read from register
3. ALU performs SUB and PC+4 is added to sign extended offset
4. Zero output from ALU used to set PC



Additional multiplexor on top to choose between jump target or either the branch target or the default PC+4

JUMP CONTROL SIGNAL controls this multiplexor

JUMP target obtained by shifting lower 26 bits left by 2 bits, then concatenating PC+4 upper 4 bits

0----------

For lecture

Good exam questions

Add jalr rs,rd 0 rs 0 rd 0 9

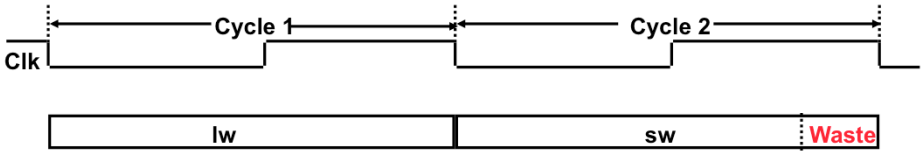
jump to instr whose addr is in rs and save addr of next inst (PC+4) in rd

Add the PowerPC addressing modes of update addressing and indexed addressing (will have to expand the RegFile to be three read port and two write port)

Add andi, ori, addi - have to have both a signextend and a zeroextend and choose between the two, will have to augment the ALUop encoding (since can't get the op information out of the funct bits as with R-type)

Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instruction
  - especially problematic for more complex instructions like floating point multiply



- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

but

- Is simple and easy to understand

In the Single Cycle implementation, the cycle time is set to accommodate the longest instruction,

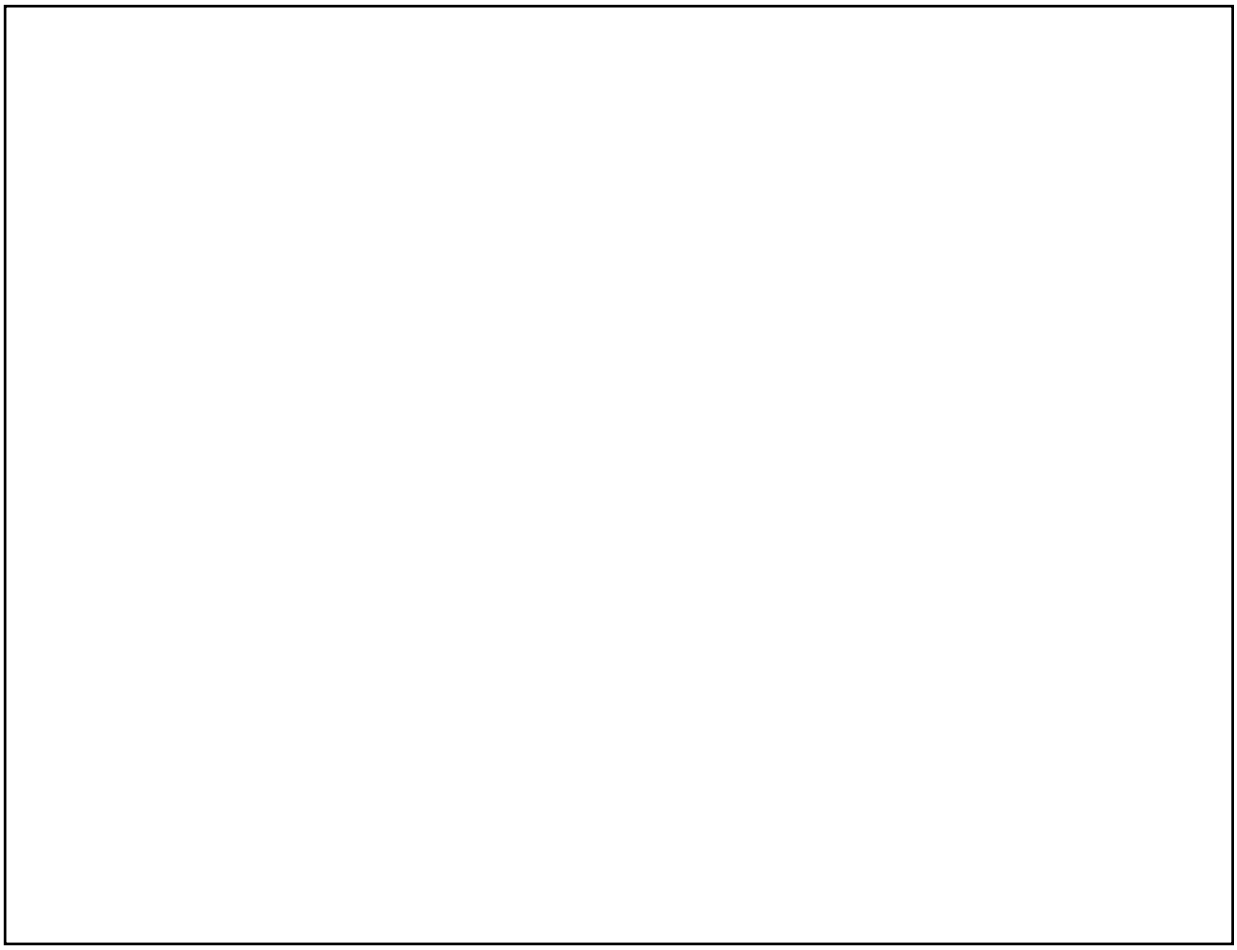
the Load instruction (because it uses 5 functional units in series: intruct memory, register file, ALU, data memory and register file).

Since the cycle time has to be long enough for the load instruction, it is too long for the store instruction so the last part of the cycle here is wasted.

Multicycle Datapath Approach

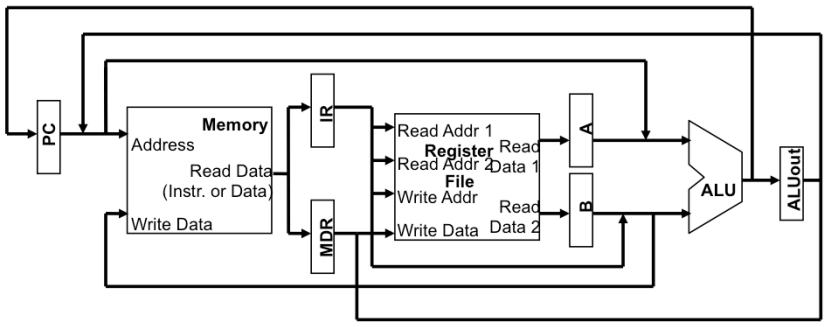
- Let an instruction take more than 1 clock cycle to complete
  - Break up instructions into steps where each *step* takes a cycle while trying to
    - balance the amount of work to be done in each step
    - restrict each cycle to use only one major functional unit
  - Not every instruction takes the *same* number of clock cycles

- In addition to faster clock rates, multicycle allows functional units that can be used more than once per instruction as long as they are used on *different* clock cycles, as a result
  - only need one memory – but only one memory access per cycle
  - need only one ALU/adder – but only one ALU operation per cycle



Multicycle Datapath Approach, con't

- At the end of a cycle
  - Store values needed in a later cycle by the current instruction in an internal register (not visible to the programmer). All (except IR) hold data only between a pair of adjacent clock cycles (no write control signal needed)



IR – Instruction Register
MDR – Memory Data Register
A, B – regfile read data registers
ALUout – ALU output register

- Data used by subsequent instructions are stored in programmer visible registers (i.e., register file, PC, or memory)

Have to add multiplexors in front of several of the functional unit inputs because the functional units are shared by different instruction cycles.

Reading/writing to

any of the internal registers or the PC occurs (quickly) at the end of a clock cycle

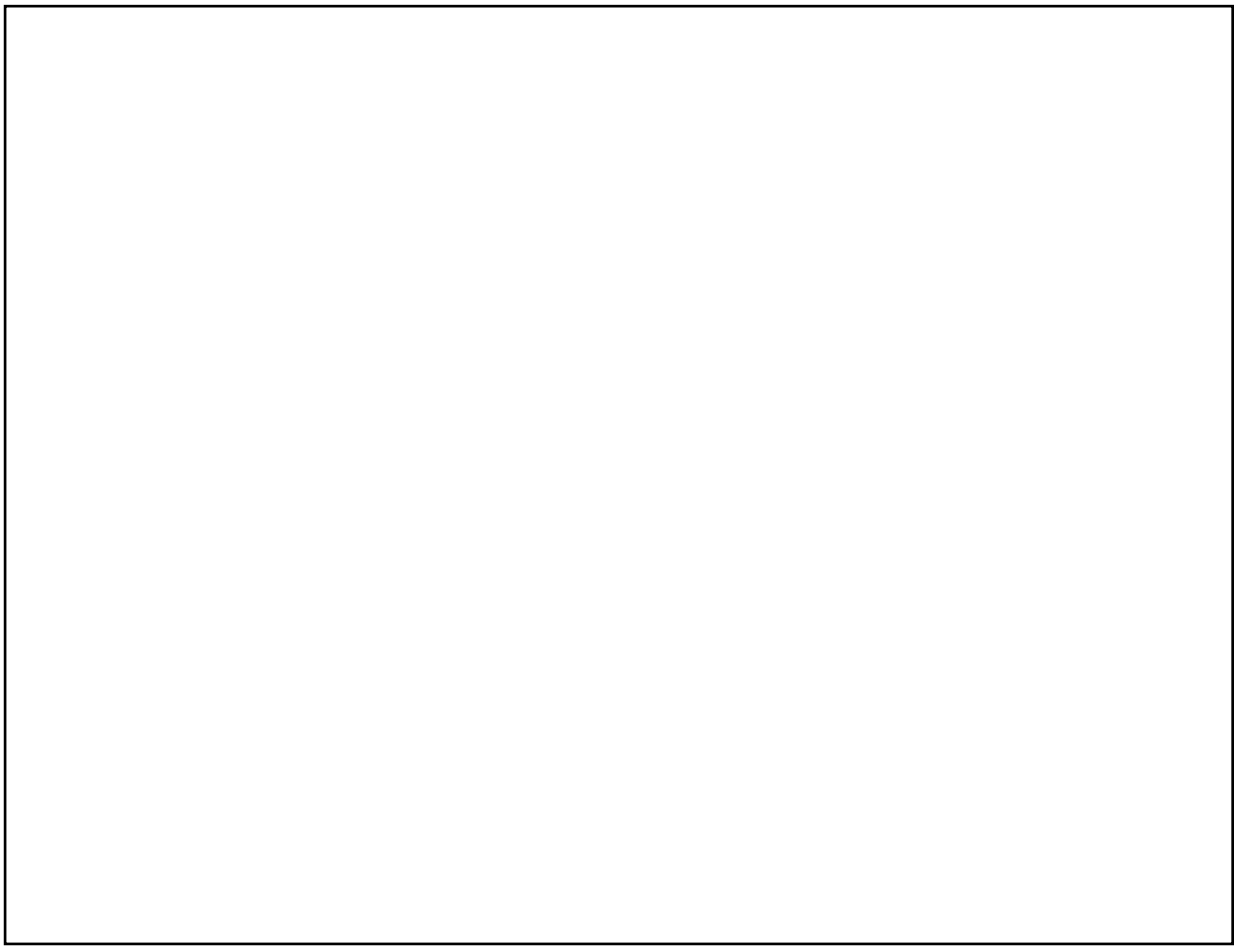
reading/writing to the register file takes ~50% of a clock cycle since it has additional control and access overhead (reading can be done in parallel with decode)

Instructions from ISA perspective

- Consider each instruction from perspective of ISA.
- Example:
  - The **add** instruction changes a register.
  - Register specified by bits 15:11 of instruction.
  - Instruction specified by the PC.
  - New value is the sum (“op”) of two registers.
  - Registers specified by bits 25:21 and 20:16 of the instruction

```
Reg[Memory[PC][15:11]] <-
   Reg[Memory[PC][25:21]] op Reg[Memory[PC][20:16]]
```

  - In order to accomplish this we must break up the instruction.
    (kind of like introducing variables when programming)



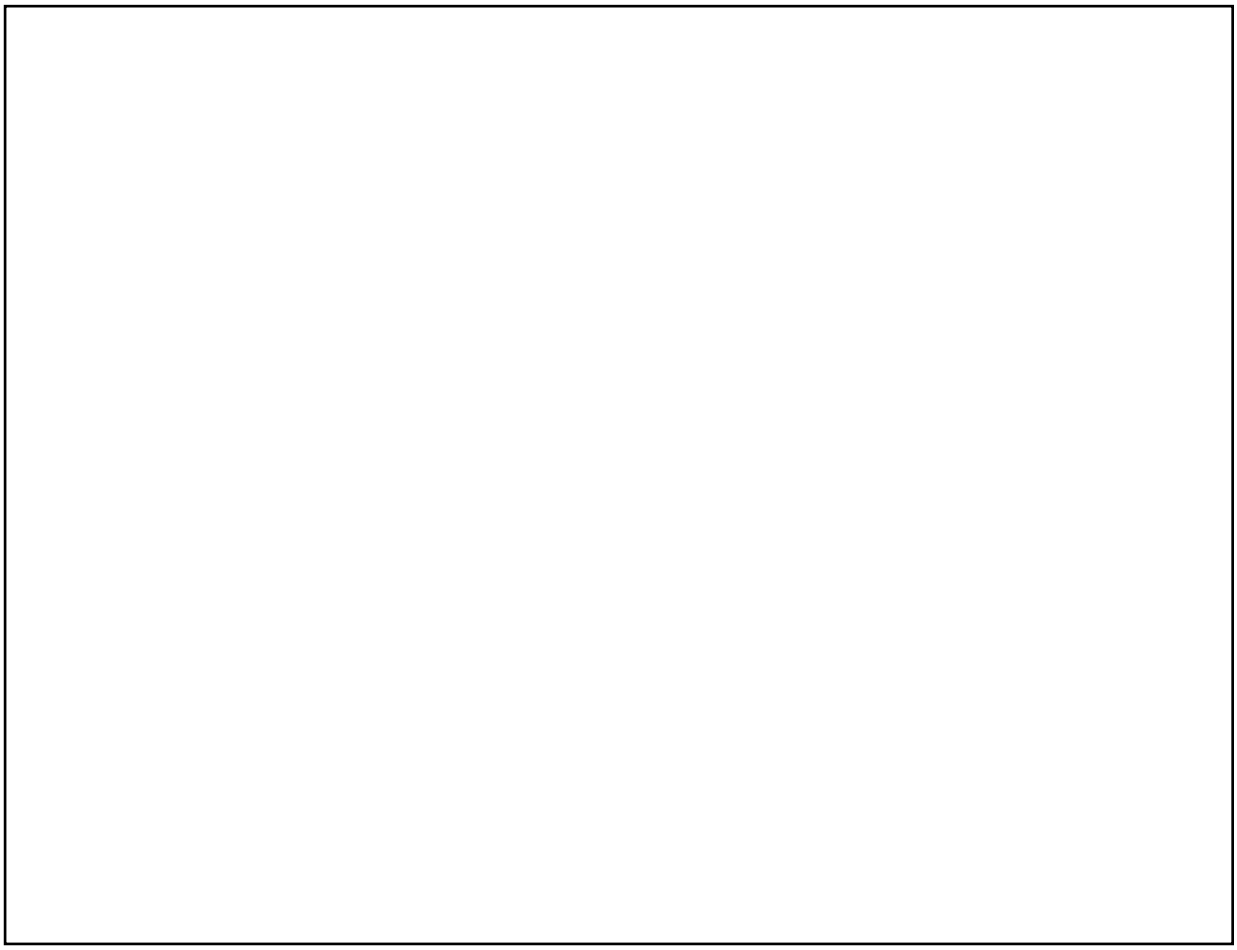
Breaking down an instruction

- ISA definition of arithmetic:

```
Reg[Memory[PC][15:11]] <-
    Reg[Memory[PC][25:21]] op Reg[Memory[PC][20:16]]
```

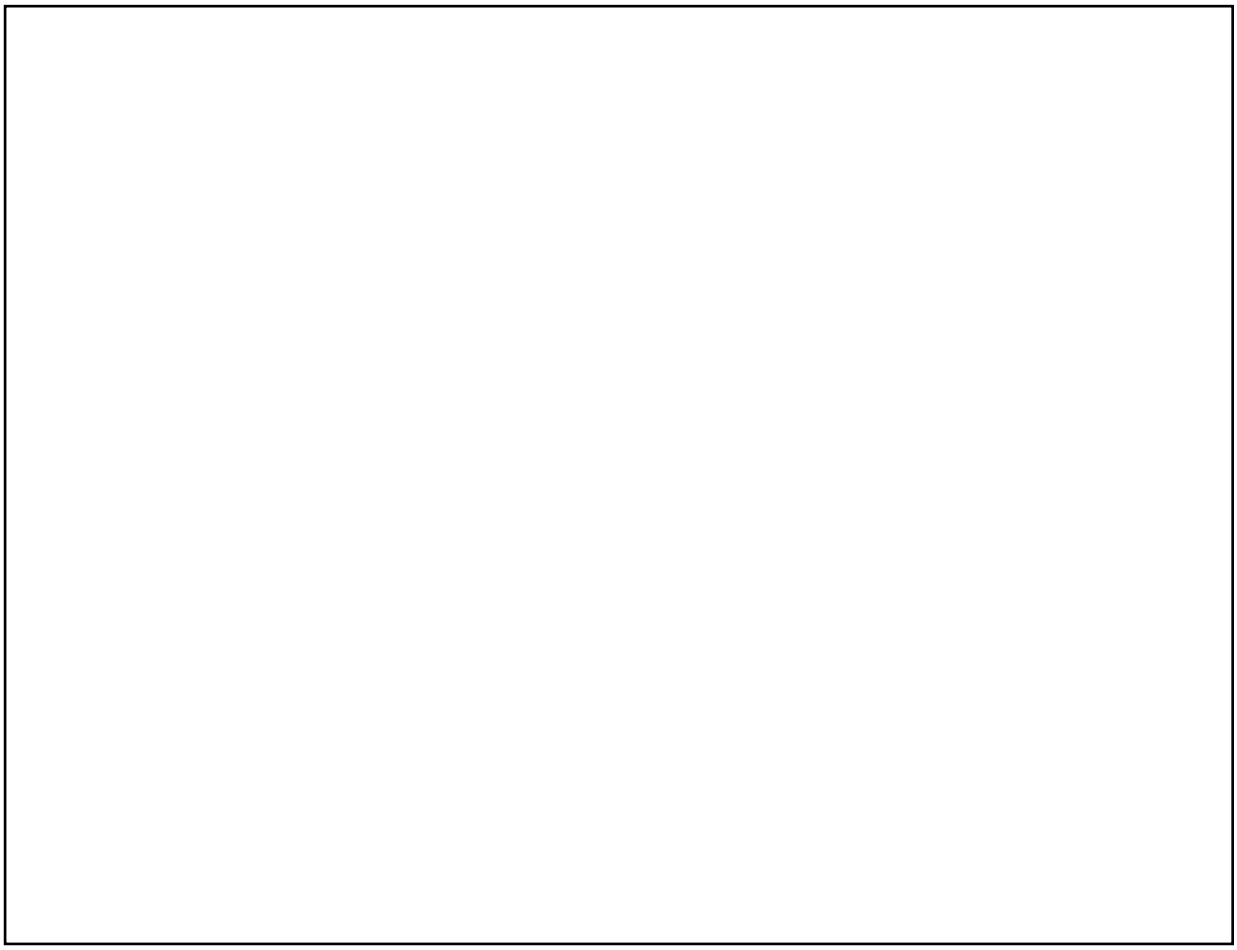
- Could break down to:
  - `IR <- Memory[PC]`
  - `A <- Reg[IR[25:21]]`
  - `B <- Reg[IR[20:16]]`
  - `ALUOut <- A op B`
  - `Reg[IR[15:11]] <- ALUOut`

- We forgot an important part of the definition of arithmetic!
  - `PC <- PC + 4`



Idea behind multicycle approach

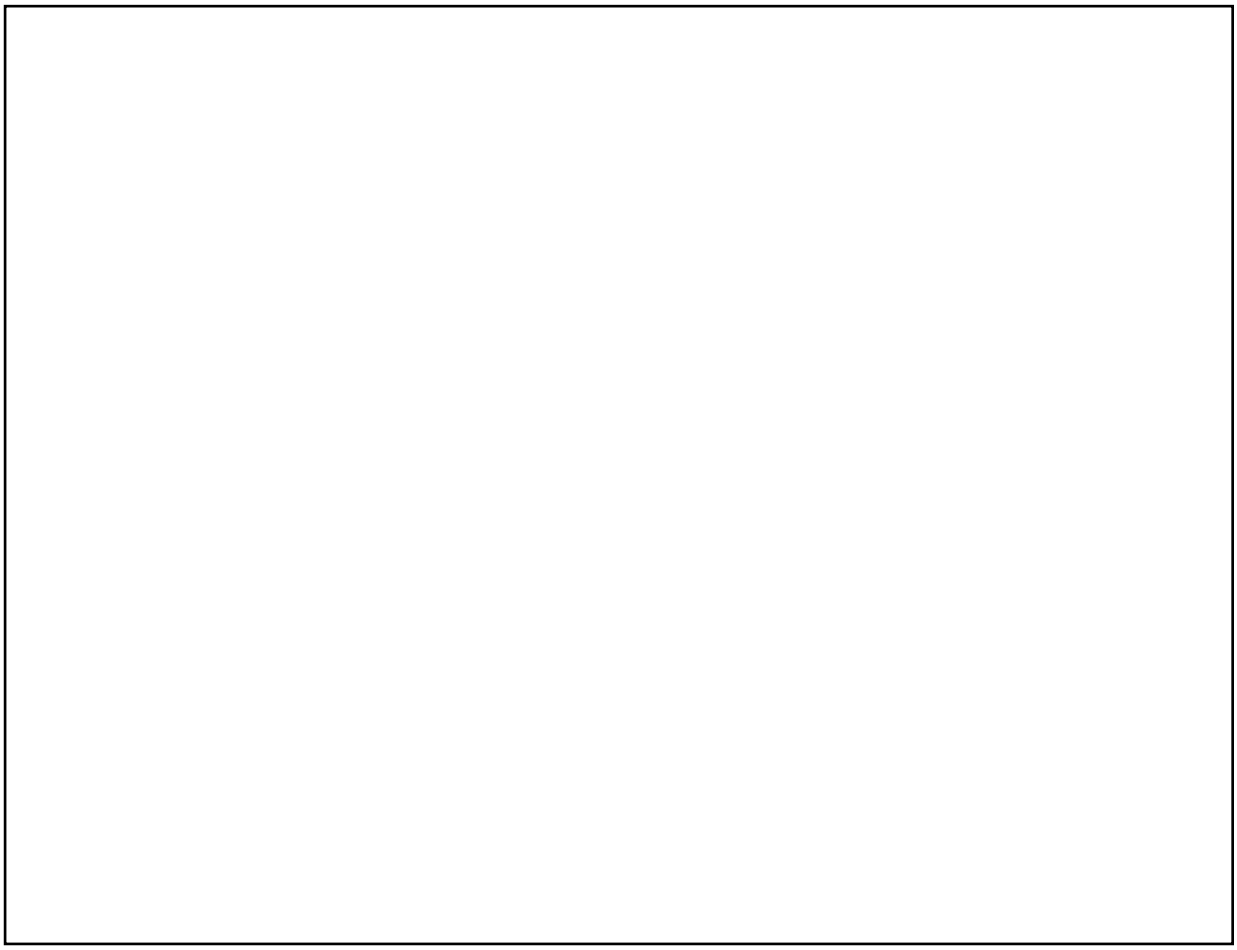
- We define each instruction from the ISA perspective (do this!)
- Break it down into steps following our rule that data flows through at most one major functional unit (e.g., balance work across steps)
- Introduce new registers as needed (e.g, A, B, ALUOut, MDR, etc.)
- Finally try and pack as much work into each step
  (avoid unnecessary cycles)
  while also trying to share steps where possible
  (minimizes control, helps to simplify solution)



Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!



Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

```
IR <= Memory[PC];
PC <= PC + 4;
```

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

Advantage: ALU wird jetzt nicht gebraucht, kann also für +4 verwendet werden.

No use of ALU now, so it can be used for +4 and also PC is immediately ready to be used at next cycle

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

```
A <= Reg[IR[25:21]];

B <= Reg[IR[20:16]];

ALUOut <= PC + (sign-extend(IR[15:0]) << 2);
```

- We aren't setting any control lines based on the instruction type
  (we are busy "decoding" it in our control logic)

Register Transfer Language

Step 3 (instruction dependent)

- ALU is performing one of three functions, based on instruction type

- Memory Reference:

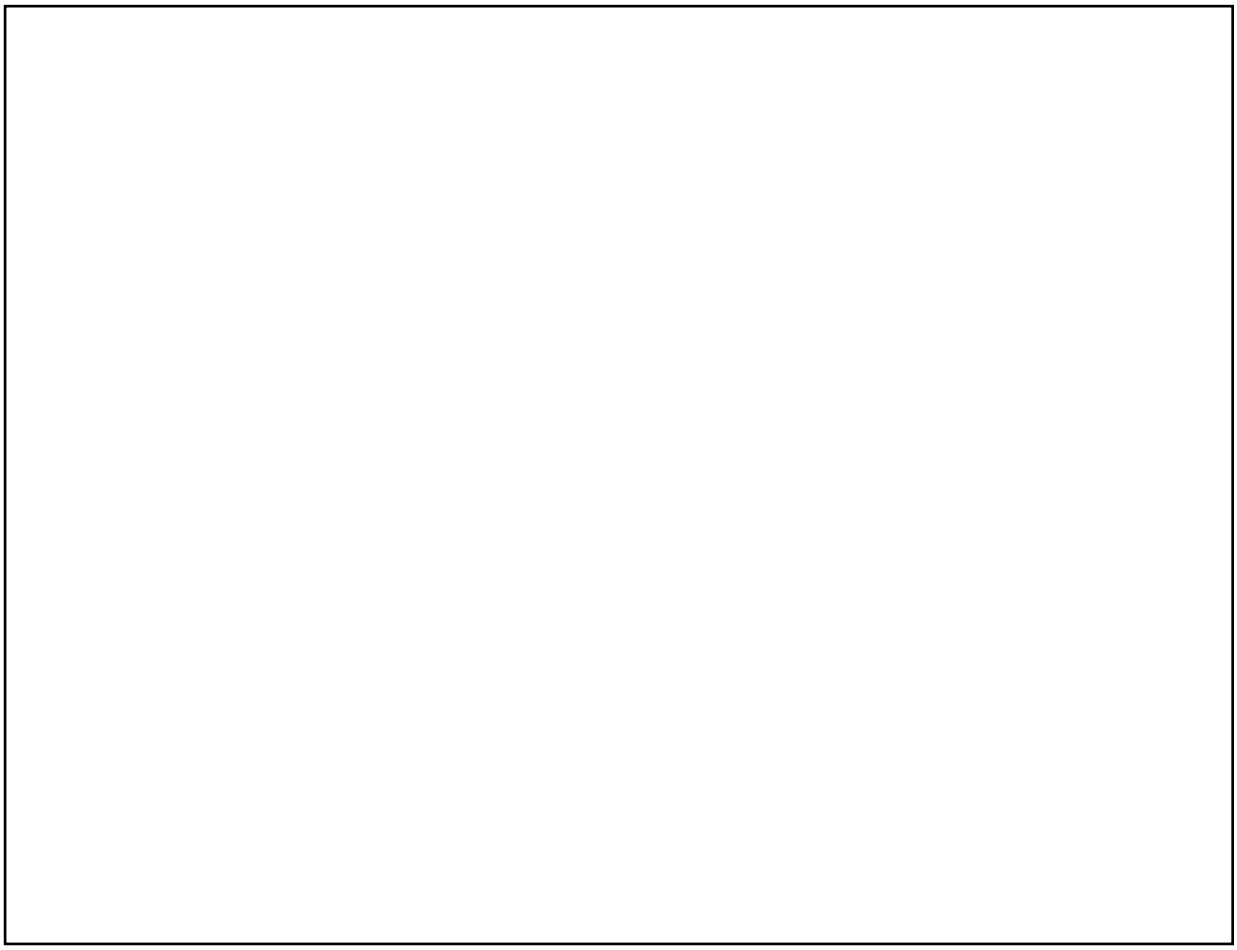
```
ALUOut <= A + sign-extend(IR[15:0]);
```

- R-type:

```
ALUOut <= A op B;
```

- Branch:

```
if (A==B) PC <= ALUOut;
```

Step 4 (R-type or memory-access)

- Loads and stores access memory

```
MDR <= Memory[ALUOut];
     or
Memory[ALUOut] <= B;
```

- R-type instructions finish

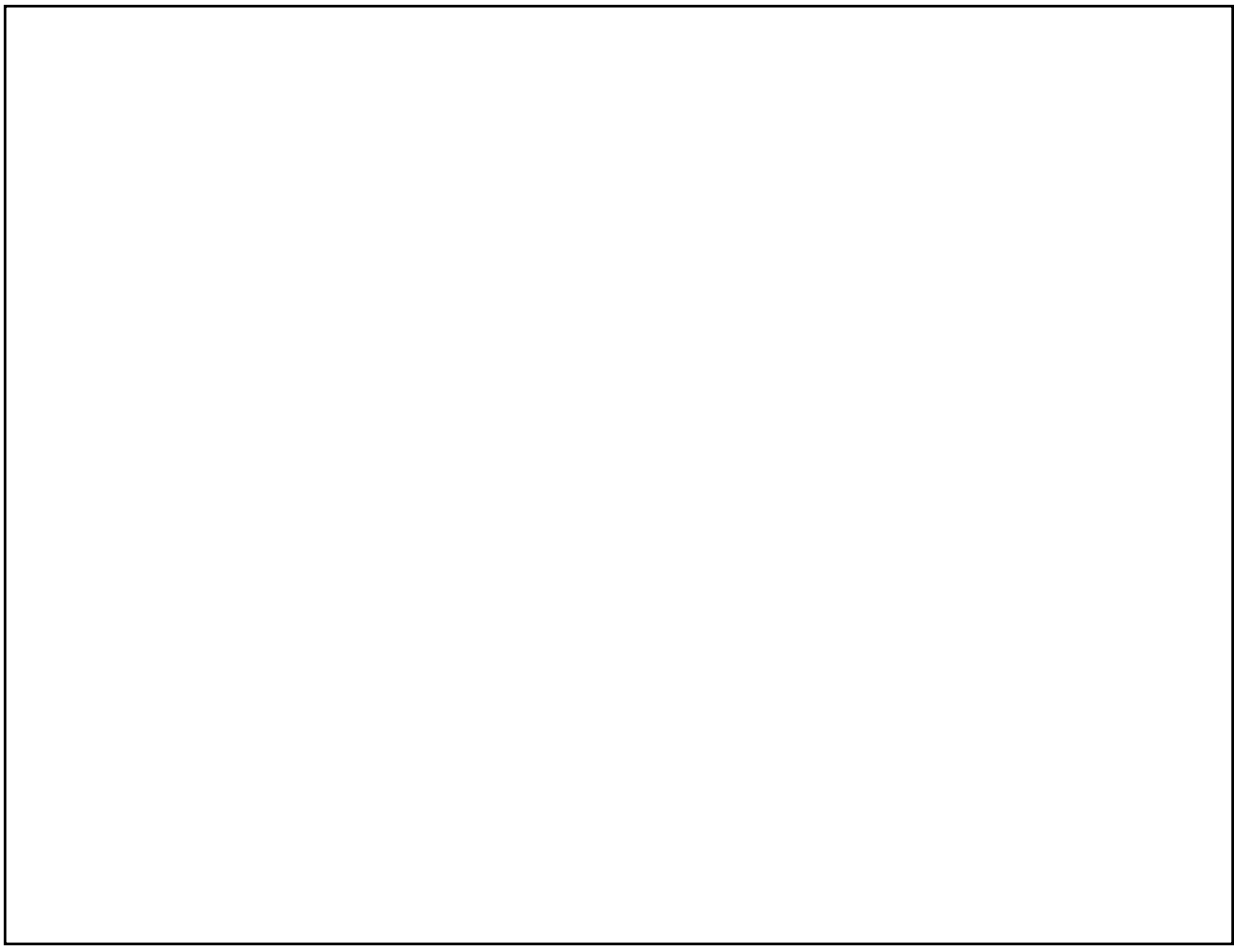
```
Reg[IR[15:11]] <= ALUOut;
```


The write actually takes place at the end of the cycle on the edge

Write-back step

- `Reg[IR[20:16]] <= MDR;`

Only Load instruction needs this cycle.



Summary:

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	If (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]),2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

Ladebefehle 5

Speicherbefehle 4

ALU Befehle 4

Verzweigungen 3

Sprünge 3

Simple Questions

- How many cycles will it take to execute this code?

```
          lw $t2, 0($t3)
          lw $t3, 4($t3)
          beq $t2, $t3, Label #assume not
          add $t5, $t2, $t3
          sw $t5, 8($t3)
Label:    ...
```

- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of `$t2` and `$t3` take place?

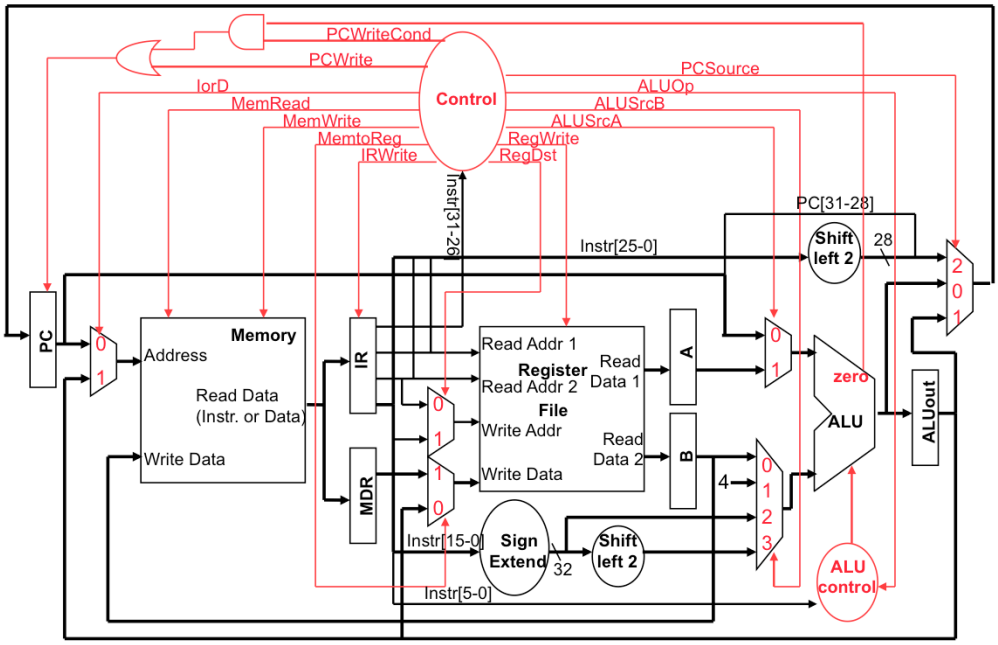
5 + 5 + 3 + 4 + 4 = 21

8: Berechnen der Speicheradresse von welcher die Daten gelesen werden sollen.

(computing the memory address from which data should be read)

Addition in Cycle 16

The Multicycle Datapath with Control Signals



IorD 0: Befehl wird gelesen 1: ALUOut Adressiert Speicher, Load oder Store

MemToReg 0: Daten von ALUOut 1: Daten von MDR

IRWrite 0: - 1: Speicher wird in IR geschrieben (braucht es da IR nicht in jedem Zyklus neu geschrieben wird)

RegDst 0: Zielregister rt [20-16] 1: Zielregister ist rt [15-11]

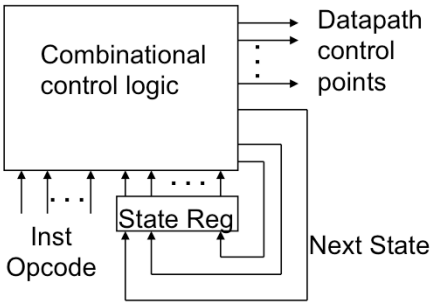
AluSrcA 0: PC 1: Reg A

AluSrcB 0: Reg B 1: Konst 4 für PC 2: IR[15-0] signed extended für speicherzugriff oder immediate arithm. Op. 3: IR[15-0] signed extended shift left um 2 bit für branches.

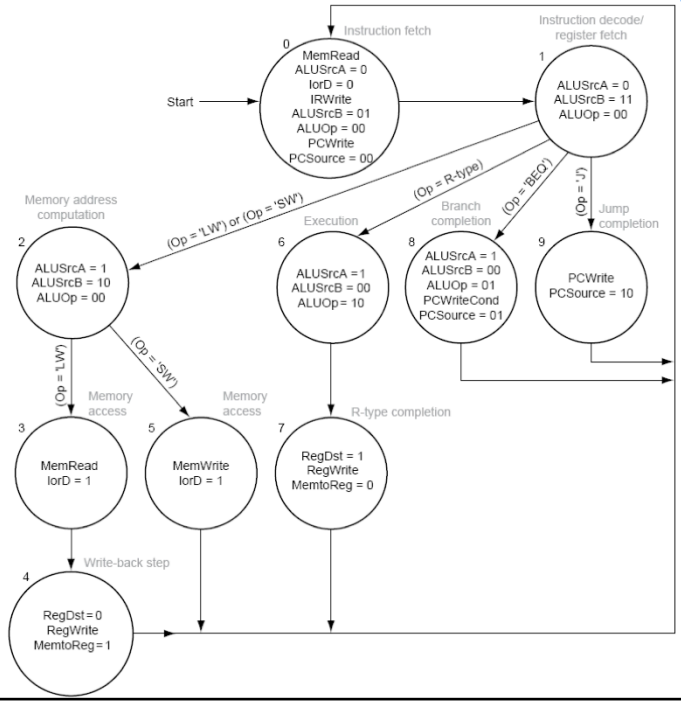
PCSrc 0: +4 1: branch 2: Jump

Multicycle Control Unit

- Multicycle datapath control signals are not determined solely by the bits in the instruction
  - e.g., op code bits tell what operation the ALU should be doing, but *not* what instruction cycle is to be done next
- Must use a finite state machine (FSM) for control
  - a set of states (current state stored in State Register)
  - next state function (determined by current state and the input)
  - output function (determined by current state and the input)

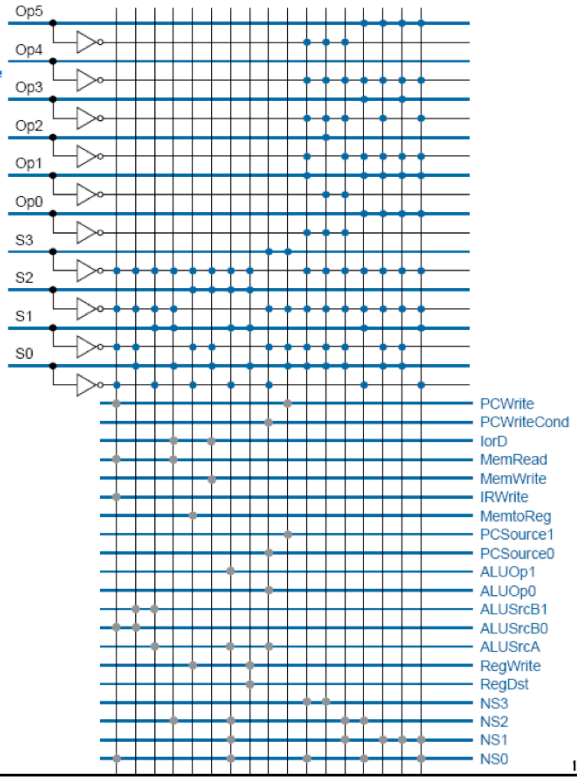


Multicycle Control Unit: Finite State Machine



Control Unit: PLA

PLA implementation of the finite state machine for the multicycle control unit

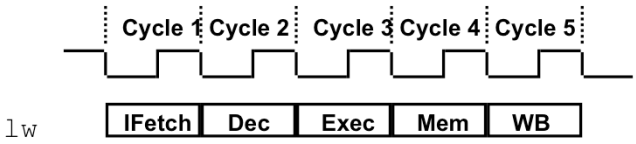


Programmed Logic Array

Each output is the logical OR or one or more minterms

Each minterm is a logical AND of one or more inputs

The Five Steps of the Load Instruction



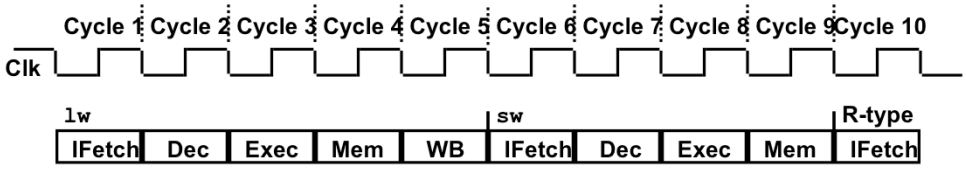
- IFetch: Instruction Fetch and Update PC
- Dec: Instruction Decode, Register Read, Sign Extend Offset
- Exec: Execute R-type; Calculate Memory Address; Branch Comparison; Branch and Jump Completion
- Mem: Memory Read; Memory Write Completion; R-type Completion (RegFile write)
- WB: Memory Read Completion (RegFile write)

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

As shown here, each of these five steps will take one clock cycle to complete.

Multicycle Advantages & Disadvantages

- Uses the clock cycle efficiently – the clock cycle is timed to accommodate the slowest instruction step

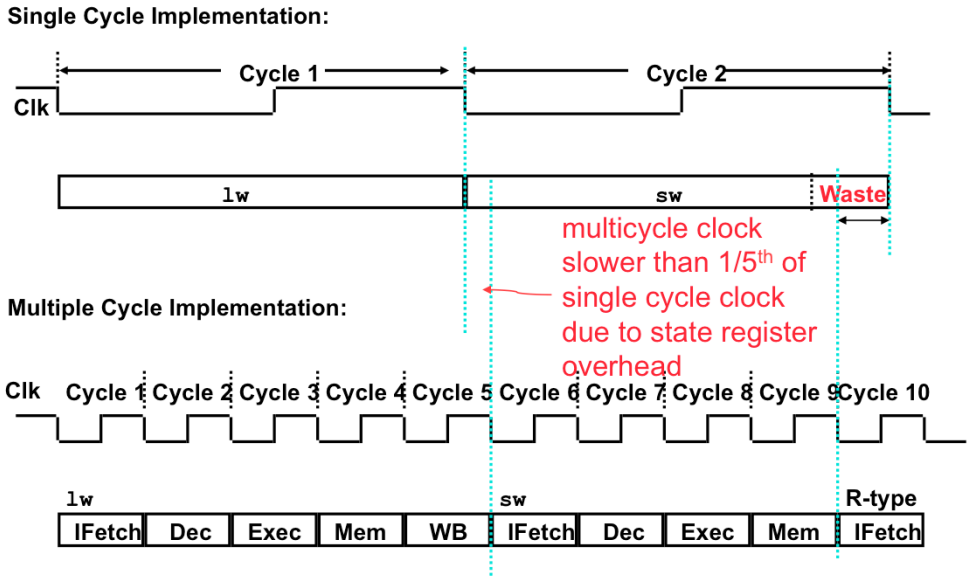


- Multicycle implementations allow functional units to be used more than once per instruction as long as they are used on different clock cycles

but

- Requires additional internal state registers, more muxes, and more complicated (FSM) control

Single Cycle vs. Multiple Cycle Timing



Here are the timing diagrams showing the differences between the single cycle and multiple cycle.

In the multiple clock cycle implementation, we cannot start executing the store until Cycle 6 because we must wait for the load instruction to complete.

Similarly, we cannot start the execution of the R-type instruction until the store instruction has completed its execution in Cycle 9.

In the Single Cycle implementation, the cycle time is set to accommodate the longest instruction, the Load instruction.

Consequently, the cycle time for the Single Cycle implementation can be five times longer than the multiple cycle implementation.