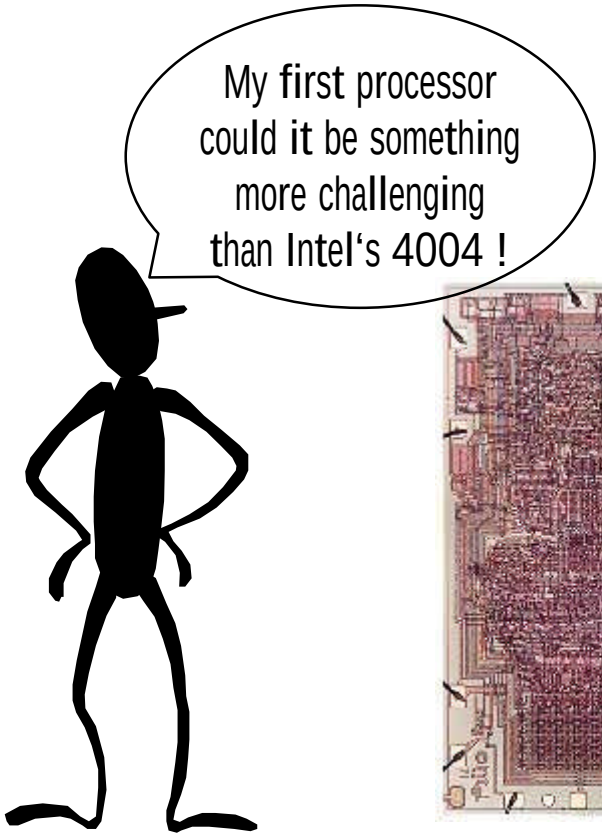
RISC Processor Design

RISC Theory
(Course No 580, by Dr. Marcel Jacomet)



1971: first microprocessor

Goal: You are able to understand the role of performance, the machine language as well as basic arithmetic for computers. You know the principles of the data-path and control path of a RISC processor and are able to enhance the performance with pipelining – and last but not least, you managed to design your first RISC processor.

Outline

Part I: RISC Theory

- Computer Organization & Design: The hardware / software interface, David A. Patterson & John L. Hennessy, Morgan Kaufmann Publishers Inc. (ISBN 1-55860-491-X)
  - Computer Abstraction and Terminology (chap 1)
  - The Role of Performance (chap 2)
  - Instructions: Language of the Machine (chap 3)
  - Arithmetic for Computers (chap 4)
  - The Processor: Datapath and Control (chap 5)
  - Enhancing Performance with Pipelining (chap 6)

Part II: MyRISC Project

- ASIP Meister IP-Core Tool (several universities from Japan) www.ed-meister.org
  - Design of MyRISC Instruction Set
  - Design of MyRISC Architecture
  - Simulation with Assembler Code Program
  - Realization of MyRISC Processor into FPGA
  - Optional: Discussion for MyRISC C/Java Compiler & OP System

- This course is based on the above textbook and also uses most of the provided slides from the authors

Computer Abstraction and Terminology #1

- C program compiled into assembly language and then assembled into binary machine language
- Abstraction:
  - Delving into depths reveals more details
  - An abstraction omits unneeded detail, helps us cope with complexity

High-level Language Program (in C)

```
Swap (int v[], int k
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1]= temp;
}
```

C complier

Assembly Language Program (for MIPS)

```
Swap:
     muli $2, $5,4
     add  $2, $4,$2
     lw   $15, 0($2)
     lw   $16, 4($2)
     sw   $16, 0($2)
     sw   $15, 4($2)
     jr   $31
```

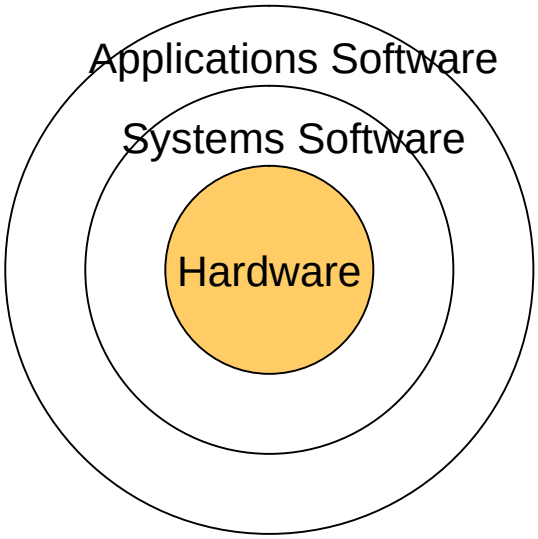
Assembler

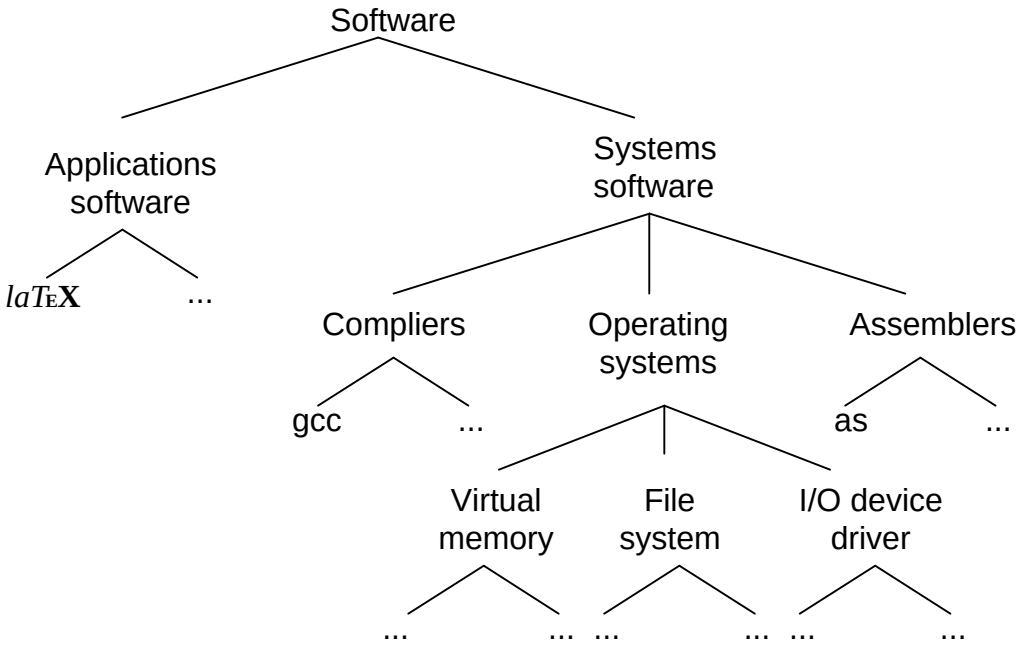
Binary Machine Language Program (for MIPS)

```
00000000101000010000000000011000
00000000100011110000110000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Computer Abstraction and Terminology #2

- instruction set architecture
  - Interface between low-level software and hardware
  - Standardizes instructions, machine language bit patterns
  - Advantage: different implementation of the same architecture
  - Disadvantage: sometimes prevents us from new innovations





Performance

- Measure, report and summarize
- Make intelligent choices
- See through the marketing hype
- Key to understand the organizational motivation

- Why is some hardware better than others for different programs?
- What factors of system performance are hardware related? (do we need a new machine or new operating system?)
- How does the machine's instruction set affect performance?

Airplane performance

Airplane	Passengers	Range (mi)	Speed (mph)
Boeing 737-100	101	630	598
Boeing 747	470	4150	610
BAC/Sud Concorde	132	4000	1350
Douglas DC-8-50	146	8720	544

- Which of these airplanes has the best performance
  - How much faster is the concorde compared to the 747
  - How much bigger is the 747 compared to the DC-8

Computer performance

- Response time (latency)
  - How long does it take for my job to run ?
  - How long does it take to execute a job ?
  - How long must I wait for the database query ?
- Throughput
  - How many jobs can the machine run at once ?
  - What is the average execution rate ?
  - How much work is getting done ?
- If we upgrade a machine with a new processor what do we increase ?
- If we add a new machine to the lab what do we increase ?

Execution time

- Elapsed time
  - Counts everything (disk and memory access, I/O, etc)
  - A useful number but often not good for comparison purposes
- CPU time
  - Doesn't count I/O or time spend running other programs
  - Can be broken up into system time and user time
- Our focus: user CPU time
  - Time spend executing the lines that are „in" our program

Book's definition of performance

- For some program running on machine „X“
- $\text{performance}_x = 1 / \text{execution time}_x$

- „X“ is n times faster than „Y“
- $\text{performance}_x / \text{performance}_y = n$

- Problem:
  - Machine A runs a program in 20 sec
  - Machine B runs same program in 25 sec

Clock cycles

- Instead of reporting execution time in seconds, we often use cycles

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

- How to improve the performance ?

- Different instructions take different amount of time on different machines
  - Multiplication takes more than one addition
  - Floating point operations take longer than integer ones
  - Accessing memory takes longer than accesing registers

- Important: changing the cycle time often changes the number of cycles required for various instructions (see later)

Cycles per instruction (CPI)

- Vocabulary
  - Cycle time (second per cycle)
  - Clock rate (cycles per second)
  - CPI (cycles per instruction, an average value)
  - A floating point intensive application might have a higher CPI
  - MIPS (millions of instructions per second)
  - This would be higher for a program using simple instructions
- Performance is determined by execution time
- Do any of the other variables equal performance
  - # of cycles to execute a program
  - # of instructions in a program
  - # of cycles per second
  - Average # of cycles per instruction
  - Average # of instructions per second
- Common pitfall: thinking one of the variables is indicative of the performance when it really isn‘t

$$\text{Time} ? \frac{\text{instructions}}{\text{program}} ? \frac{\text{clock cycles}}{\text{instruction}} ? \frac{\text{seconds}}{\text{clock cycle}}$$

CPI example

Example (difficulty: easy): Suppose we have two implementations of the same instruction set architecture (ISA). For some program

- Machine A has a clock cycle time of 10 ns and a CPI of 2.0
- Machine B has a clock cycle time of 20 ns and a CPI of 1.2

What machine is faster for this program and by what factor?

If two machines have the same ISA, which of our quantities (clock rate, CPI, execution time, # of instructions, MIPS) will always be identical?

of instructions example

Example (difficulty: easy): A compiler designer is trying to decide between two code sequences for a particular machine. Based on the hardware implementation, there are three different classes of instructions: Class A, Class B, and Class C, and they require one, two, and three cycles respectively.

- The first code sequence has 5 instructions: 2 of A, 1 of B, and 2 of C
- The second sequence has 6 instructions: 4 of A, 1 of B, and 1 of C

Which sequence will be faster? How much?
What is the CPI for each sequence?

MIPS example

Example (difficulty: easy): Two different compilers are being tested for a 100 MHz machine with three different classes of instructions: Class A, Class B, and Class C, which require one, two, and three cycles respectively. Both compilers are used to produce code for a large piece of software.

- The first compiler's code uses 5 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.
- The second compiler's code uses 10 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

Which sequence will be faster according to MIPS? Which sequence will be faster according to execution time?

Admahl's law example

Example (difficulty: easy): Suppose we enhance a machine making all floating-point instructions run five times faster. If the execution time of some benchmark before the floating-point enhancement is 10 seconds, what will the speedup be if half of the 10 seconds is spent executing floating-point instructions?

Example (difficulty: easy): We are looking for a benchmark to show off the new floating-point unit described above, and want the overall benchmark to show a speedup of 3. One benchmark we are considering runs for 100 seconds with the old floating-point hardware. How much of the execution time would floating-point instructions have to account for in this program in order to yield our desired speedup on this benchmark?

Benchmarks

- Performance best determined by running a real application
  - Use programs typical of expected workload
  - Or, typical of expected class of applications
    e.g., compilers/editors, scientific applications, graphics, etc.
- Small benchmarks
  - nice for architects and designers
  - easy to standardize
  - can be abused
- SPEC (System Performance Evaluation Cooperative)
  - companies have agreed on a set of real program and inputs
  - can still be abused (Intel's "other" bug)
  - valuable indicator of performance (and compiler technology)
  - Spec89, spec95,

Remember

- Performance is specific to a particular program/s
  - Total execution time is a consistent summary of performance
- For a given architecture performance increases come from:
  - increases in clock rate (without adverse CPI affects)
  - improvements in processor organization that lower CPI
  - compiler enhancements that lower CPI and/or instruction count
- Pitfall: expecting improvement in one aspect of a machine's performance to affect the total performance
- You should not always believe everything you read! Read carefully!

Instructions

- Instructions: language of the machine
- More primitive than higher level languages
  - e.g., no sophisticated control flow
- Very restrictive
  - e.g., MIPS Arithmetic Instructions
- We'll be working with the MIPS instruction set architecture

- ***Design goals: maximize performance and minimize cost, reduce design time***

MIPS Arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

- Example:
  - C Code **A = B + C**
  - MIPS Code **`add $s0, $s1, $s2`**
    (associated with variables by compiler)

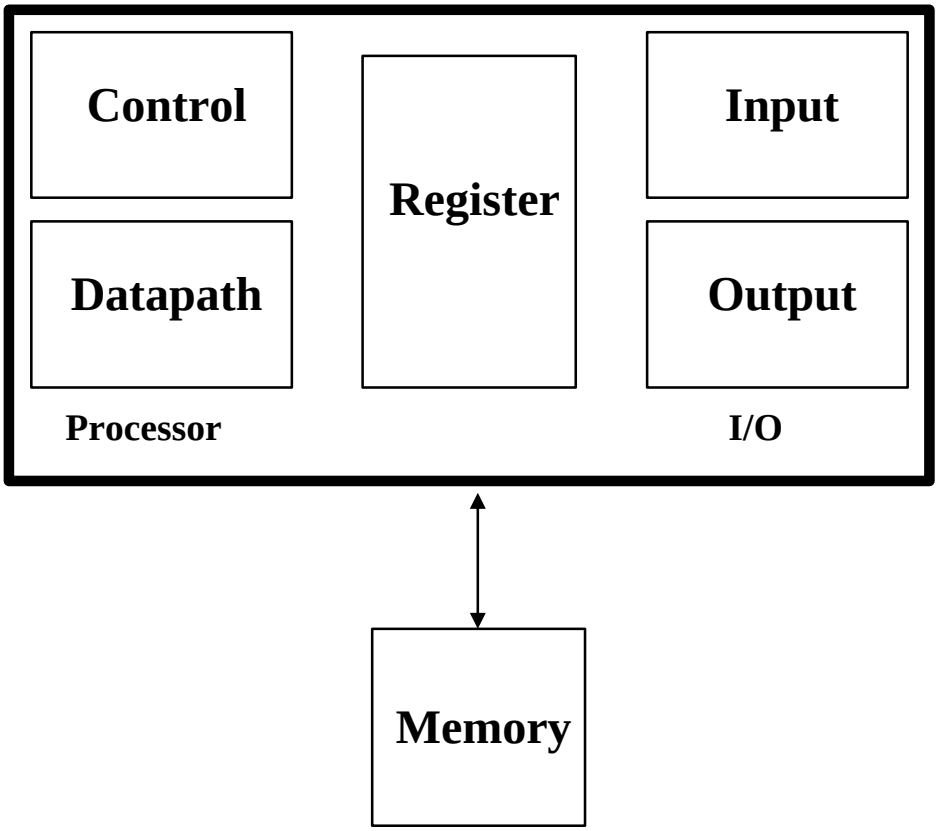
- Design Principle: simplicity favors regularity
- Of course this complicates some things...

  - C Code **A = B + C + D**
    E = F – A
  - MIPS Code **`add $t0, $s1, $s2`**
    - **`add $s0, $t0, $s3`**
      `sub $s4, $s5, $s0`

- Operands must be registers, only 32 registers provided
- Design Principle: smaller is faster

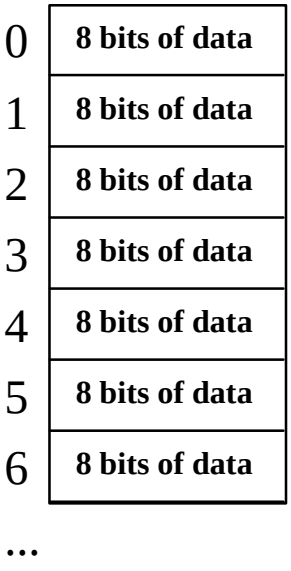
Registers vs. Memory

- Arithmetic instructions operands must be registers
- Compiler associates variables with registers
- What about programs with lots of variables

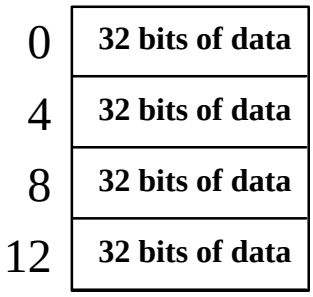


Memory organization

- Viewed as a large, single-dimension array, with an address
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory



- Bytes are nice, but most data items use larger "words“
- For MIPS, a word is 32 bits or 4 bytes



Registers hold 32 bits of data

Load & Store Instructions

- Load and store instructions
- Example:
  - C Code `A[8] = h + A[8];`
  - MIPS Code `lw $t0, 32($s3)`
    - `add $t0, $s2, $t0`
    - `sw $t0, 32($s3)`

    (store word has destination last)
- MIPS
  - loading words but addressing bytes
  - arithmetic on registers only

instruction	meaning
`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`
`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`
`lw $s1,100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1,100($s2)`	`Memory[$s2+100] = $s1`

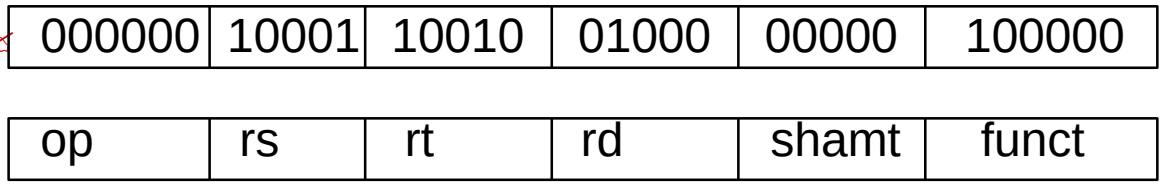
Machine Language (R-Type format)

- Instructions, like registers and words of data, are also 32 bits long

  Example: **`add $t0, $s1, $s2`**

  registers have numbers, **`$t0=8, $s1=17, $s2=18`**

- Instruction Format (R-Type)



- 6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

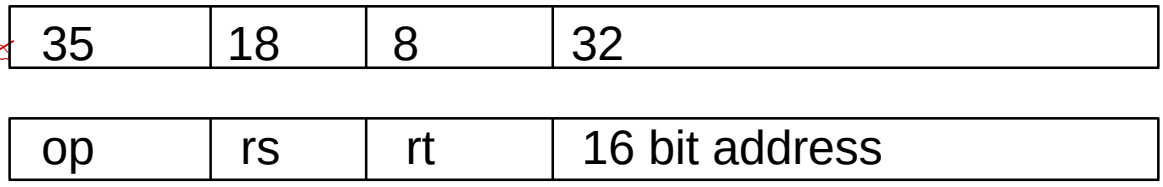
  - **`op`**: basic operation of the instruction, traditionally called opcode
  - **`rs`**: the first register source operand
  - **`rt`**: the second register source operand
  - **`rd`**: the register destination operand, it gets the result of the operation
  - **`shamt`**: shift amount
  - **`funct`**: This field selects the specific variant of the operation in the op field, and sometimes called function code

Machine Language (I-Type format)

- Consider the load-word and store-word instructions
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise

  Example: **`lw $t0, 32($s2)`**

- Instruction Format (I-Type)



- 6 bits 5 bits 5 bits 16 bits

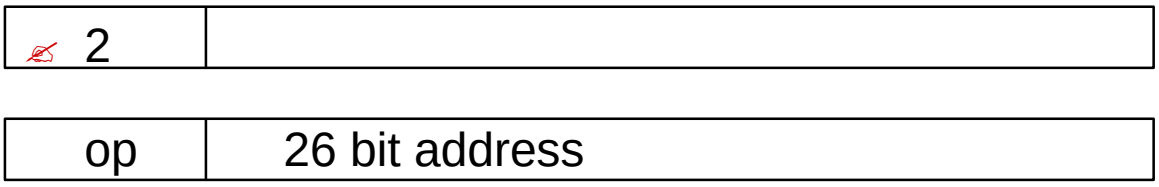
Machine Language (J-Type format)

- Decision making instructions
  - alter the control flow
  - i.e., change the "next" instruction to be executed
  - conditional branch instructions (**bne, beq**, etc)
  - unconditional branch instructions (j)

  Example:

C code	MIPS code
if (i!=j)	**beq $s4, $s5, lab1**
h=i+1;	**add $s3, $s4, $s5**
else	**j lab2**
h=i-j;	**lab1:sub $s3,$s4,$s5**
	lab2: …

- Instruction Format (J-Type)



- 6 bits 26 bits

Policy of Use Convention

- Decision making instructions
  - We have: **`beq, bne`**, what about branch-if-less-than?
  - New instruction

  Example:

  C code **`if ($s1<$s2) $t1=1;else $t1=0;`**

  MIPS code **`slt $t0, $s1, $s2`**

- Can use this instruction to build
  - **`blt $s1, $s2, Label`**
  - can now build general control structures
- Note that the assembler needs a register to do this
  - there are policy of use conventions for registers

Name	Reg Num	Usage
`$zero`	0	the constant value 0
`$v0 - $v1`	2 – 3	values for results & expression eval
`$a0 - $a3`	4 – 7	arguments
`$t0 - $t7`	8 – 15	temporaries
`$s0 - $s7`	16 – 23	saved
`$t8 - $t9`	24 – 25	more temporaries
`$qp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return pointer

Constants

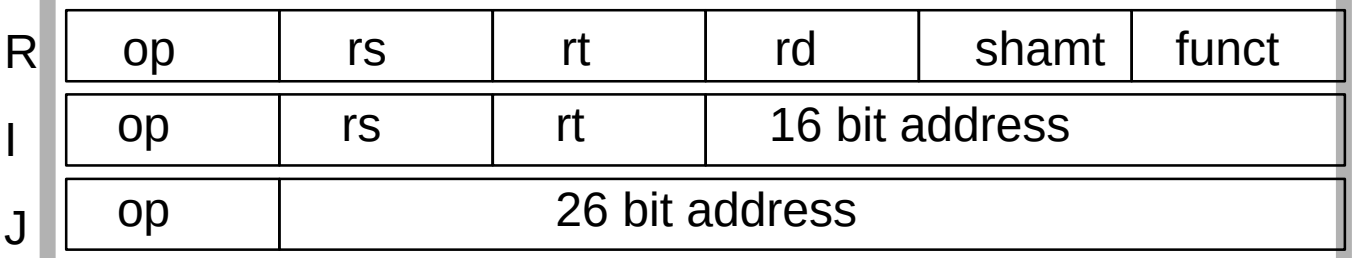
- Small constants are used quite frequently (50% of operands)
  - **eg: A = A + 5;**
- Solutions? Why not?
  - put 'typical constants' in memory and load them
  - create hard-wired registers (like $zero) for constants like one
- MIPS instructions
  - **`addi $29, $29, 4`**
  - **`slti $8, $18, 10`**
- We'd like to be able to load a 32 bit constant into a register
  - Must use two instructions, new "load upper immediate" instruction
  - **`lui $t0, 1010101010101010`**
  - Then must get the lower order bits right, i.e.,
  - **`ori $t0, $t0, 1010101010101010`**

Assembly vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - e.g., **`move $t0, $t1`** exists only in assembly
  - would be implemented using **`add $t0,$t1,$zero`**
- When considering performance you should count real instructions

Overview of MIPS Instruction Formats

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats



- rely on compiler to achieve performance
- help compiler where we can

To Summarize: Instructions

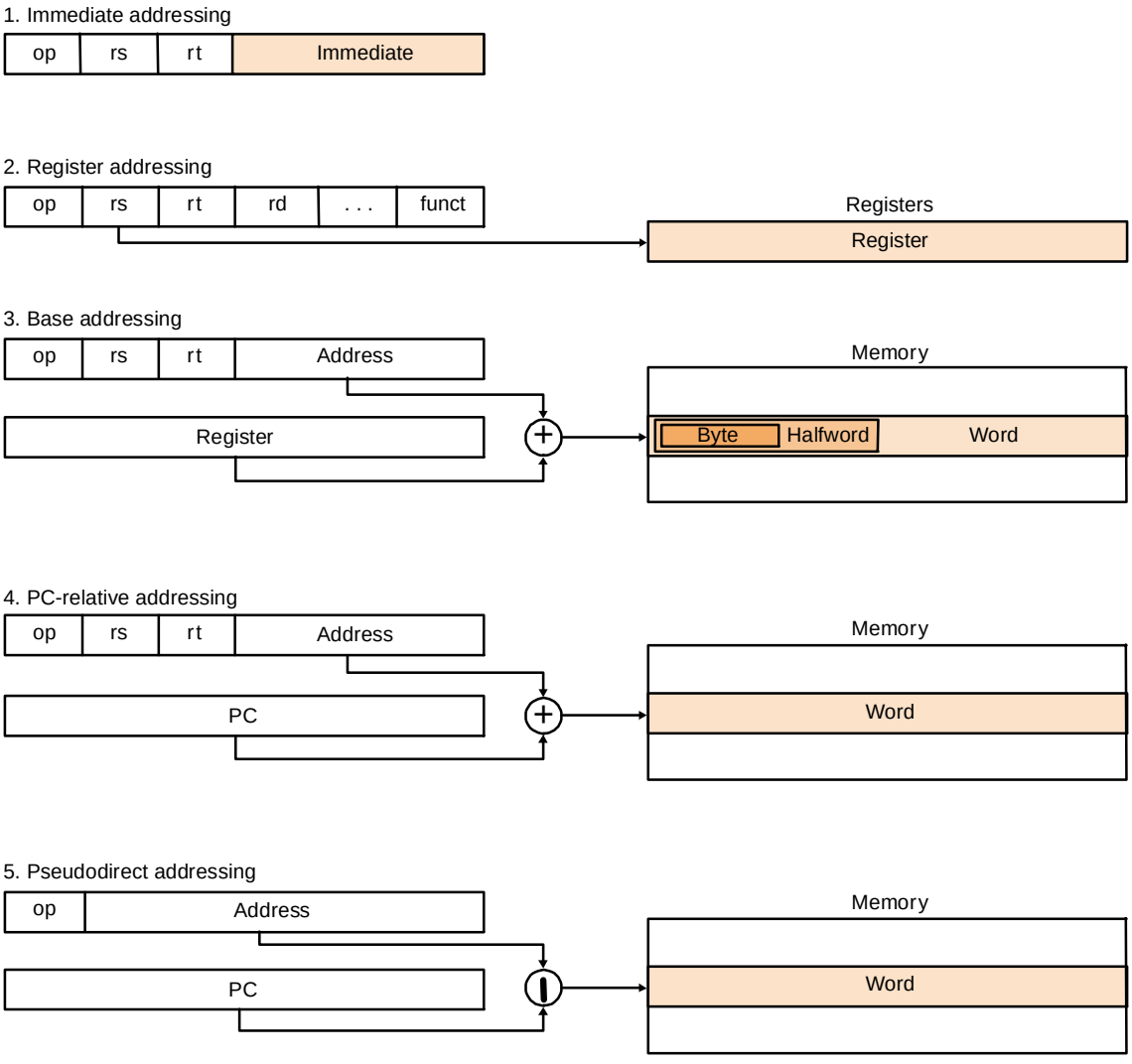
MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1 =` Memory[`$s2` + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = `$s1`	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1 =` Memory[`$s2` + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = `$s1`	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

To Summarize: Addressing Modes



Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast
- Instruction set architecture:
  - a very important abstraction indeed!

Compiling #1

- Compiling a C assignment using registers

C-code

F = (g + h) – (i + j);

MIPS-code

```
add $t0, $s1, $s2  # register $t0 contains g+h
add $t1, $s3, $s4  # register $t1 contains i+j
sub $s0, $t0, $t1  # register $s0 contains (g+h)-(i+j)
```

Compiling #2

- Compiling using load and store

C-code
A[12] = h + A[8];

MIPS-code

```
lw $t0, 32($s3)     # temp register $t0 gets A[8]
add $t0, $s2, $t0   # temp register $t0 gets h+A[8]
sw $t0, 48($s3)     # stores h+A[8] into A[12]
```

- Compiling using a variable index

C-code
g = h + A[i];

MIPS-code

```
add $t1, $s4, $s4   # temp register $t1 = 2 * i
add $t1, $t1, $t1   # temp register $t1 = 4 * i
add $t1, $t1, $s3   # $t1 = address of A[i] (4*i+$s3)
lw $t0, 0($t1)      # temp register $t0=A[i]
add $s1, $s2, $t0   # g=h+ A[i]
```

Compiling #3

- Compiling an **if** statement into a conditional branch

C-code

```
      If (i==j) go to L1;
      f = g + h;
L1:   f = f – i;
```

MIPS-code

```
      beq $s3,$s4,L1  # goto L1 if i equals j
      add $t0,$s1,$s2 # f = g + h (skipped if i equals j)
L1: sub $s0,$s0,$s3 # f=f – i (always executed)
```

- Compiling **if-then-else** into conditional branch

C-code

```
If (i==j) f = g + h; else f = g - h;
```

MIPS-code

```
       bne $s3,$s4,else # goto else if i not equal j
       add $s0,$s1,$s2  # f = g + h (skipped if i !=j)
       j exit           # goto exit
else: sub $s0,$s1,$s2  # f = g – h (skipped if i = j)
exit:
```

Compiling #4

- Compiling a loop with a variable index

C-code

```
Loop:   g = g + A[i];
        i = i + j;
        if (i != h) goto loop;
```

MIPS-code

```
loop: add $t1,$s3,$s3   # temp reg $t1 = 2 * i
      add $t1,$t1,$t1   # temp reg $t1 = 4 * i
      add $t1,$t1,$s5   # $t1 = address of A[i]
      lw $t0,0($t1)     # temp reg $t0 = A[i]
      add $s1,$s1,$t0   # g = g + A[i]
      add $s3,$s3,$s4   # i = i + j
      bne $s3,$s2,loop  # go to loop if i != h
```

Compiling #5

Compiling a while loop

C-code

```
while (save[i] == k)
        i = i + j;
```

MIPS-code

```
loop: add $t1,$s3,$s3  # temp reg $t1 = 2 * i
      add $t1,$t1,$t1  # temp reg $t1 = 4 * i
      add $t1,$t1,$s6  # $t1 = address of save[i]
      lw $t0,0($t1)    # temp reg $t0 = save[i]
      bne $t0,$s5,exit # goto exit if save[i] != k
      add $s3,$s3,$s4  # i = i + j
      j loop           # go to loop
exit:
```

(MIPS code can be optimized)

Compiling #6

- Compiling a case/switch statement

C-code

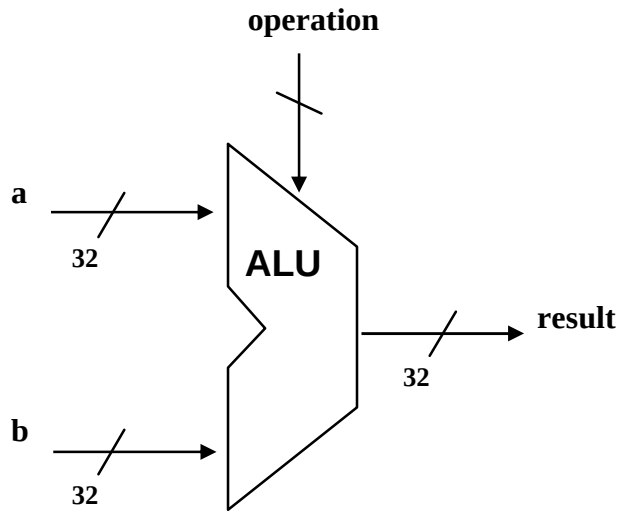
```
switch (k) {
        case 0: f = i + j;    break;
        case 1: f = g + h;    break;
        case 2: f = g - h;    break;
        case 3: f = i - j;    break;
}
```

MIPS-code

```
        slt $t3,$s5,$zero   # test if k < 0
        bne $t3,$zero,exit  # if k<0, goto exit
        slt $t3,$s5,$t2     # test if k<4
        beq $t3,$zero,exit  # if k>=4, goto exit
        add $t1,$s5,$s5     # temp reg $t1=2*k
        add $t1,$t1,$t1     # temp reg $t1=4*k
        add $t1,$t1,$t4     # $t1 = address of JumpTable[k]
        lw $t0,0($t1)       # temp reg $t0 = JumpTable[k]
        jr $t0              # jump based on reg $t0
L0:     add $s0,$s3,$s4     # k=0 so f gets i+j
        j exit              # end of this case so goto exit
L1:     add $s0,$s1,$s2     # k=1 so f gets g+h
        j exit              # end of this case so goto exit
L2:     sub $s0,$s1,$s2     # k=2 so f gets g-h
        j exit              # end of this case so goto exit
L3:     sub $s0,$s3,$s4     # k=3 so f gets i-j
exit:                       # end of switch statement
```

Arithmetic for Computers

- Where we have been
  - performance (seconds, cycles, instructions)
  - abstractions
    - instruction set architecture
    - assembly language and machine language
- What's up ahead
  - implementing the architecture



Numbers

Integers:

- unsiged numbers
- signed numbers
  - two‘s complement numbers
  - 32 bit numbers: -2‘147‘483‘648 ... 2‘147‘483‘647
  - converting to negativ numbers: invert all bits and add 1
  - converting n bit numbers in numbers with more than n bits: copy the most significant bit (sign bit) into the other bits
    - 0010 -> 0000 0010
    - 1010 -> 1111 1010
    - sign extension „lbu“ vs. „lb“

Floats:

- what

Addition & Subtraction

- just like in primary school (carry/borrow 1s)

```
   0111        0111        0110
 + 0001      - 0110      - 0101
```

- two's complement operations (easy)

```
   0111
 + 1010
 ------
   0001
```

- overflow (results too large for finit word length)

```
   0111
 + 0001
 ------
   1000
```

Overflow

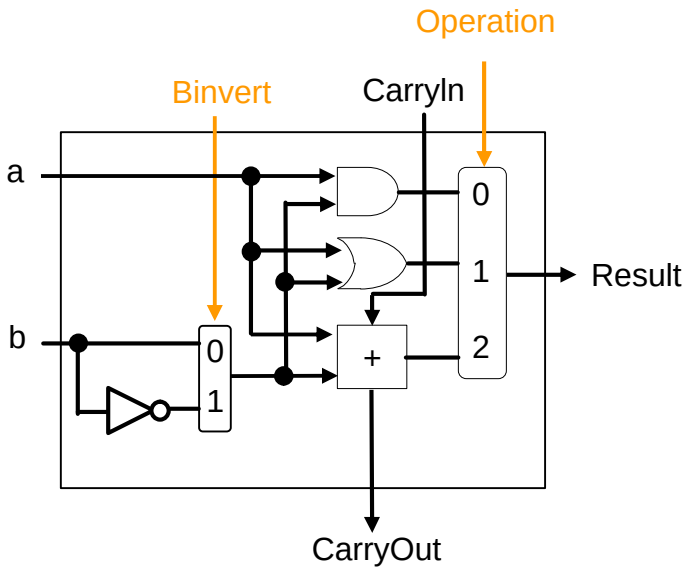
- detecting overflow
  - overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, adding two positives gives a negative
  - or, subtracting a negative from a positive gives a negative
  - or, subtracting a positive from a negative gives a positive
- consider the operation A-B, can overflow occur if A=0

- effects of overflow:
- an exception (interrupt) occurs
  - control jumps to predefined address for exception
  - interrupted address is saved for possible resumption
  - add (**`add`**), add immediate (**`addi`**), subtract (**`sub`**)
- don't always want to detect overflow
  - C ignores overflows
  - unsigned integers are commonly used for memory addresses where overflows are ignored
  - add unsigned (**`addu`**), add immediate unsigned (**`addiu`**), subtract unsigned (**`subu`**)
  - set less than unsigned (**`sltu`**), set less than immediate unsigned (**`sltiu`**)

still sign extends!

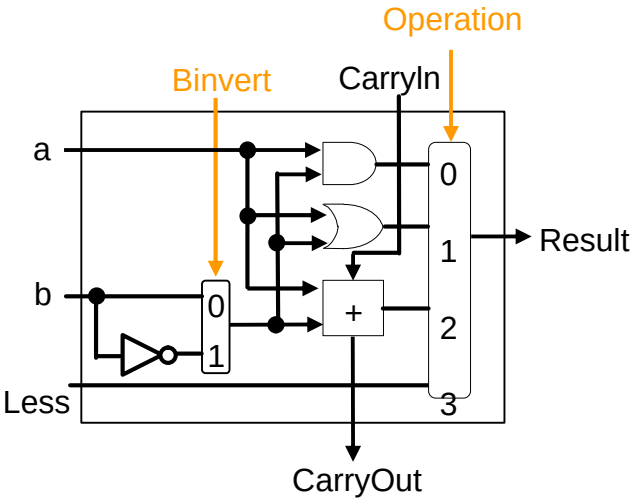
Arithmetic Logic Unit #1

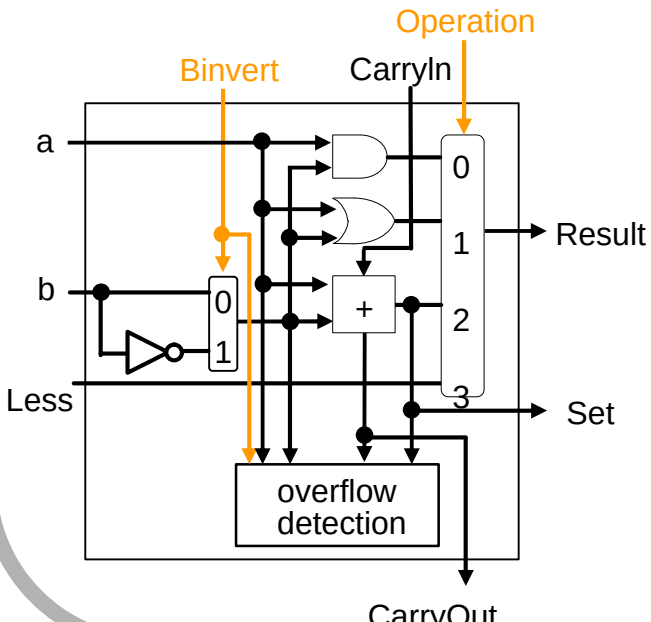
- ALU operations: **and**, **or**, **add**, **sub** (signed and unsigned)

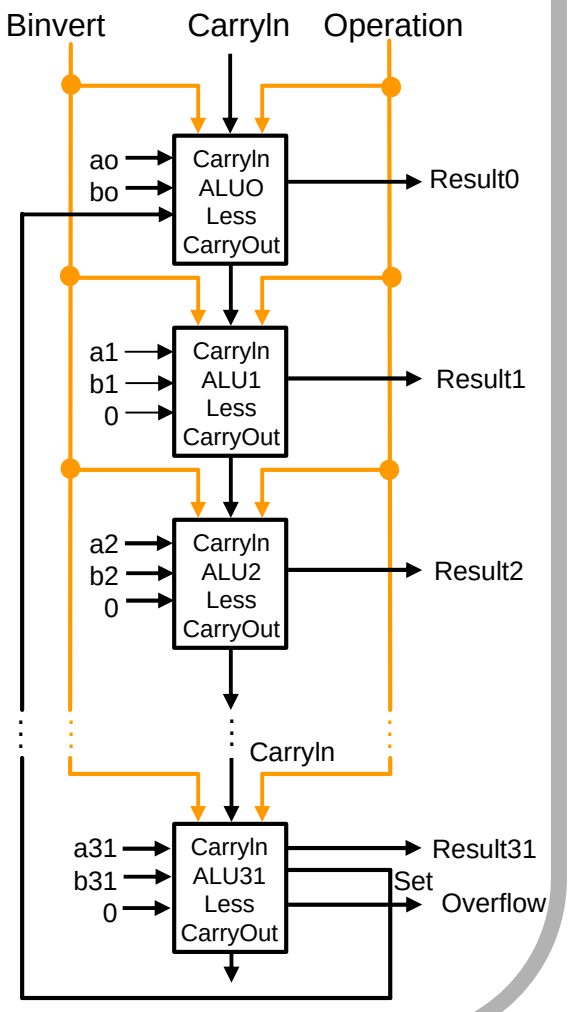


Arithmetic Logic Unit #2

- tailoring the ALU to the MIPS
  - need to support set-on-less-than **`slt`**
    - produces a 1 if rs < rt and 0 otherwise
    - use substraction (a-b) < 0 implies a < b
  - need to support test on equality **`beq`**
    - use substraction (a-b) = 0 implies a = b

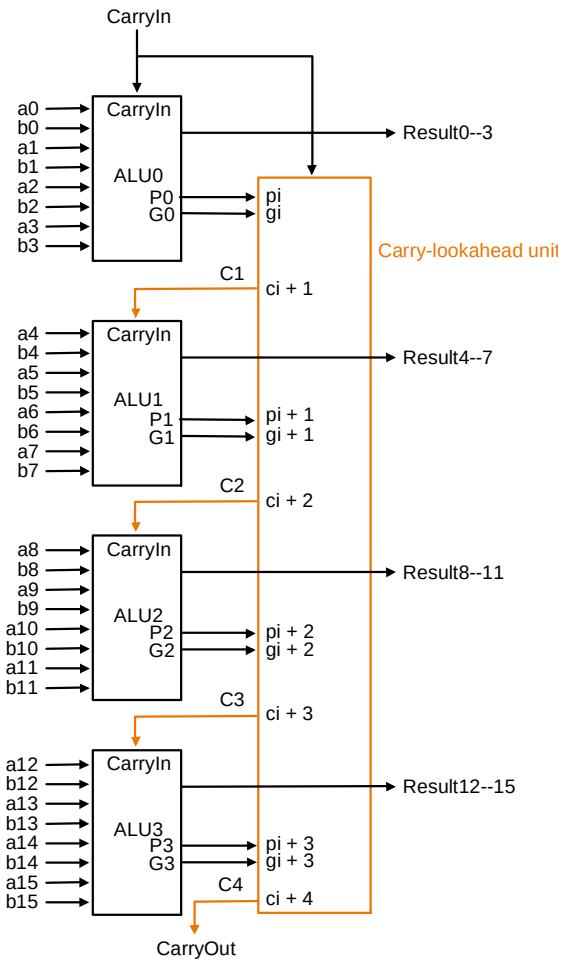






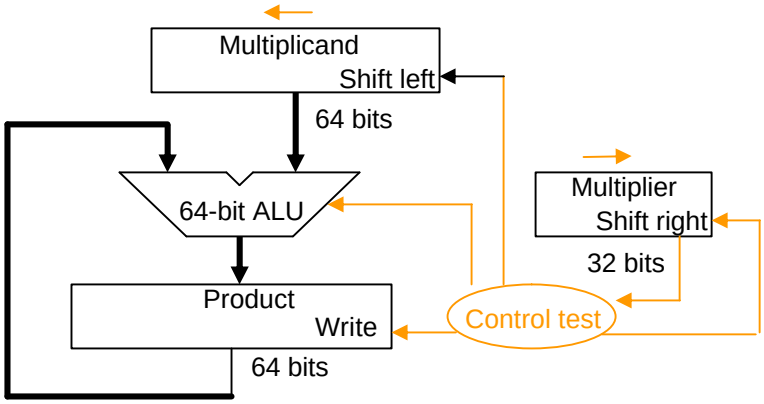
Adders

- ripple-carry adders are too slow for 32 bit
- use carry lookahead or even hierarchical carry lookahead technique (see VLSI course)

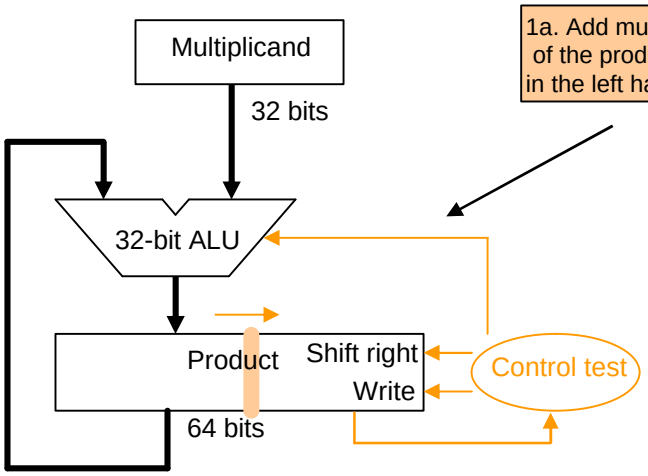


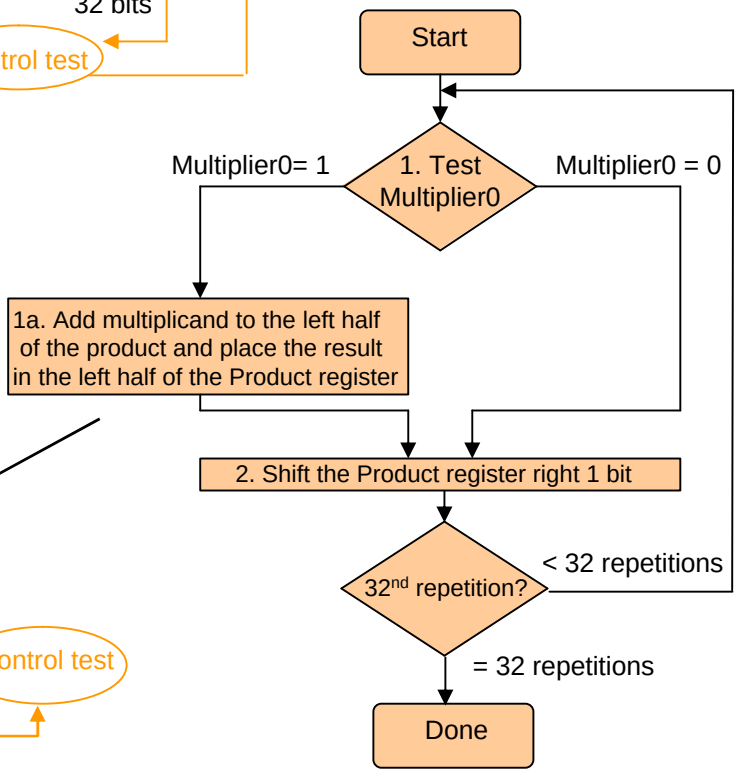
Multiplication

- multiplication is more complicated than addition:
  - implementation by shift and addition (time vs area)
  - implementation by booth recoding algorithm (see VLSI)
  - negative numbers: convert and then multiply



optimized version
(serial implementation)





- MIPS provides a separata pair or 32 bit registers containing the 64 bit product called „Hi“ and „Lo“
  - signed (**`mult`**), unsigned multiplicstion (**`multu`**)
  - MIPS pseudosinstruction are used to fetch the 32 bit integer product move from lo (**`mflo`**) and move from hi (**`mfhi`**)

Floating Point

- We need a way to represent
  - numbers with a fraction: e.g. **`5.1916`**
  - very small numbers: e.g. **`0.000000012`**
  - very large numbers: e.g: **`3.1577 x 10`^{12}**
- Representation
  - sign, exponent, significant:
  - **`(-1)`$^{\texttt{sign}}$ `x significant x 10`$^{\texttt{exponent}}$**
  - more bits for significant increases accuracy
  - more bits for exponent increases range
- IEEE 754 floating point standard
  - single precision: 8 bit exponent, 23 bit significant
  - double precision: 11 bit exponent, 52 bit significant

IEEE 754 Floating Point Standard

- leading „1“ bit of significant is implicit
- exponent is „biased“ to make sorting easier
  - all 0 is smallest exponent, all 1 is largest
  - bias 127 for single precision, 1023 for double precision

$$(-1)^{sign} \times (1+significant) \times 10^{exponent-bias}$$

- examples:
  - decimal: -0.75 = $-3/2^2$
  - binary: -0.11 = -1.1×2^{-1}
  - floating point exponent: exponent = 126 = 0111111
  - IEEE 754 single precision:
  - **10111111010000000000000000000000**

- in addition to overflow we can have underflow
- accuracy can be a big problem
  - IEEE 754 keeps to extra bits: guard and round
  - four rounding modes
  - positive divided by zero gives „infinity“
  - zero divided by zero yields „not a number“
  - other complexities

MIPS Core Instructions (Arithmetic)

instruction	example	meaning	comments
add	`add $s1,$s2,$s3`	`$s1=$s2+$s3`	overflow detected
add immediate	`addi $s1,$s2,100`	`$s1=$s2+100`	+const, overfl detected
add unsigned	`addu $s1,$s2,$s3`	`$s1=$s2+$s3`	overflow undetected
add immediate unsigned	`addiu $s1,$s2,100`	`$s1=$s2+100`	+ constant overflow undetected
sub	`sub $s1,$s2,$s3`	`$s1=$s2-$s3`	overflow detected
sub unsigned	`subu $s1,$s2,$s3`	`$s1=$s2-$s3`	overflow undetected
move from coproc reg	`mfc0 $s1,$epc`	`$s1=$epc`	used to copy exception PC plus other spec reg
multiply	`mult $s2,$s3`	`Hi,Lo=$s2x$s3`	64 bit signed in Hi,Lo
multiply unsig	`multu $s2,$s3`	`Hi,Lo=$s2x$s3`	64 bit unsig in Hi,Lo
divide	`div $s2,$s3`	`Lo=$s2/$s3`	Lo = quotient
		`Hi=$s2 mod $s3`	Hi = reminder
divide unsig	`divu $s2,$s3`	`Lo=$s2/$s3`	Lo = unsigned quotient
	`Hi=$s2 mod $s3`		Hi = unsigned reminder
move from Hi	`mfhi $s1`	`$s1=Hi`	used to get copy of Hi
move from Lo	`mflo $s1`	`$s1=Lo`	used to get copy of Lo

Core Instructions (logical, data transfer)

instruction	example	meaning	comments
and	`and $s1,$s2,$s3`	`$s1=$s2&$s3`	logical AND
or	`or $s1,$s2,$s3`	`$s1=$s2¦$s3`	logical OR
andi	`andi $s1,$s2,100`	`$s1=$s2&100`	logical AND reg const
ori	`ori $s1,$s2,100`	`$s1=$s2¦100`	logical OR
shift left logic	`sll $s1,$s2,10`	`$s1=$s2<<10`	shift left by const
shift right log	`srl $s1,$s2,10`	`$s1=$s2>>10`	shift right by const
load word	`lw $s1,100($s2)`	`$s1=mem[$s2+100]`	word from mem to reg
store word	`sw $s1,100($s2)`	`mem[$s2+100]=$s1`	word from reg to mem
load byte unsig	`lbu $s1,100($s2)`	`$s1=mem[$s2+100]`	byte from mem to reg
store byte unsigned	`sbu $s1,100($s2)`	`mem[$s2+100]=$s1`	byte from reg to memory
load upper immediate	`lui $s1,100`	`$s1=100*2^16`	load const in upper 16 bits

Core Instructions (conditional branch, jump)

instruction	example	meaning	comments
branch on equal	`beq $s1,$s2,25`	`if ($s1==$s2) go to PC+4+100`	equal test; PC relative branch
branch on not equal	`bne $s1,$s2,25`	`if ($s1!=$s2) go to PC+4+100`	not equal test; PC relative branch
set on less than	`slt $s1,$s2,$s3`	`if ($s2<$s3) $s1=1 else $s1=0`	compare less than two's complement
set on less than immedi	`slti $s1,$s2,100`	`if ($s2<100) $s1=1 else $s1=0`	compare < const two's complement
set less than unsigned	`sltu $s1,$s2,$s3`	`if ($s2<$s3) $s1=1 else $s1=0`	compare less than natural numbers
set less than imm unsigned	`sltiu $s1,$s2,100`	`if ($s2<100) $s1=1 else $s1=0`	compare < const natural numbers
jump	`j 2500`	`go to 10000`	jump to target addr
jump regist	`jr $ra`	`go to $ra`	for switch,proced ret
jump and link	`jal 2500`	`$ra=PC+4,goto 10000`	for proc call

Arithmetic Core Instructions (floating point)

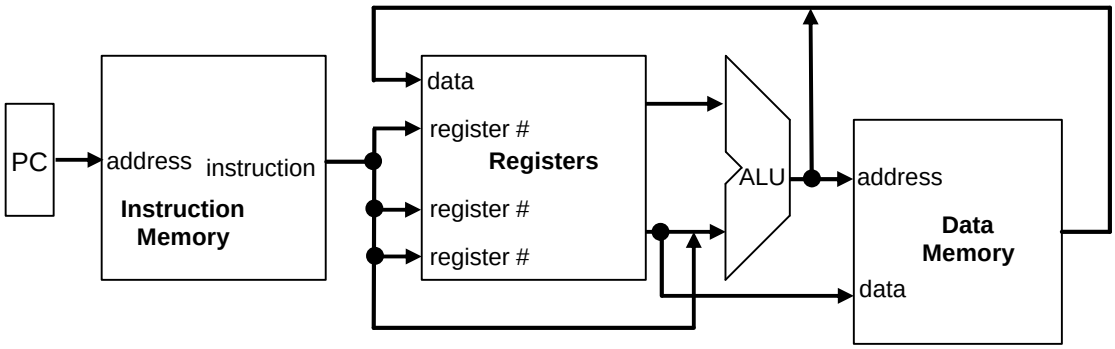
instruction	example	meaning	comments
fp add single prec	`add.s $f2,$f4,$f6`	`$f2=$f4+$f6`	floating point single
fp sub single	`sub.s $f2,$f4,$f6`	`$f2=$f4-$f6`	fp single prec sub
fp muliply sin	`mul.s $f2,$f4,$f6`	`$f2=$f4*$f6`	fp single prec mulitply
fp divide sin	`div.s $f2,$f4,$f6`	`$f2=$f4/$f6`	fp single prec divide
fp add double	`add.d $f2,$f4,$f6`	`$f2=$f4+$f6`	fp double prec add
fp sub double	`sub.d $f2,$f4,$f6`	`$f2=$f4-$f6`	fp double prec sub
fp muliply dou	`mul.d $f2,$f4,$f6`	`$f2=$f4*$f6`	fp double prec mulitply
fp divide doub	`div.d $f2,$f4,$f6`	`$f2=$f4/$f6`	fp double prec divide
load word copr	`lwc1 $f1,100($f2)`	`$f1=mem[$s2+100]`	32 bit data to fp
store word copr	`swc1 $f1,100($f2)`	`mem[$s2+100]=$f1`	32 bit data to mem
branch on fp true	`bc1t 25`	`if(cond==1)go to PC+4+100`	PC relative branch if fp condition
branch on fp false	`bc1f 25`	`if(cond==0)go to PC+4+100`	PC relative branch if fp not condition
fp compare single (eq,ne,lt,le,gt,ge)	`c.le.s $f2,$f4`	`if($f2<$f4) cond=1 else 0`	fp compare less than single prec
fp compare double (eq,ne,lt,le,gt,ge)	`c.x.d $f2,$f4`	`if($f2<$f4) cond=1 else 0`	fp compare less than double prec

Datapath and Control

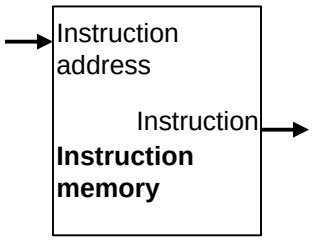
- We are ready to look at an implementation of the RISC processor (MIPS compliant)
- Simplified version to contain only:
  - memory reference instructions: **`lw,sw`**
  - arithmetic-logic instructions: **`add,sub,and,or,slt`**
  - control flow instructions: **`beq,j`**
- Generic implementation:
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do
- All instructions use the ALU after reading the register
  - why? Memory reference? Arithmetic? Control flow?
- Make the common sense fast
- Simplicity favors regularity

The Processor: Abstract View

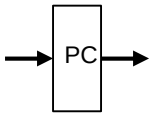
- Two types of functionl units:
  - elements that operate on data (combinatorial)
  - elements that contain state (sequential)



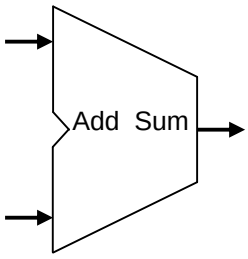
Functional Units



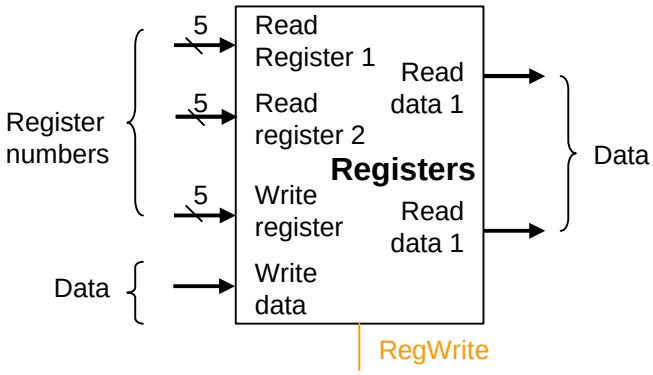
Instruction Memory



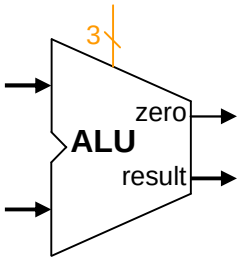
Program Counter



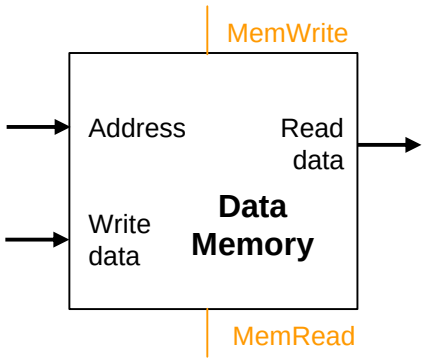
Adder



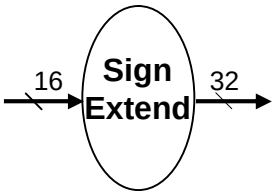
Register File



ALU

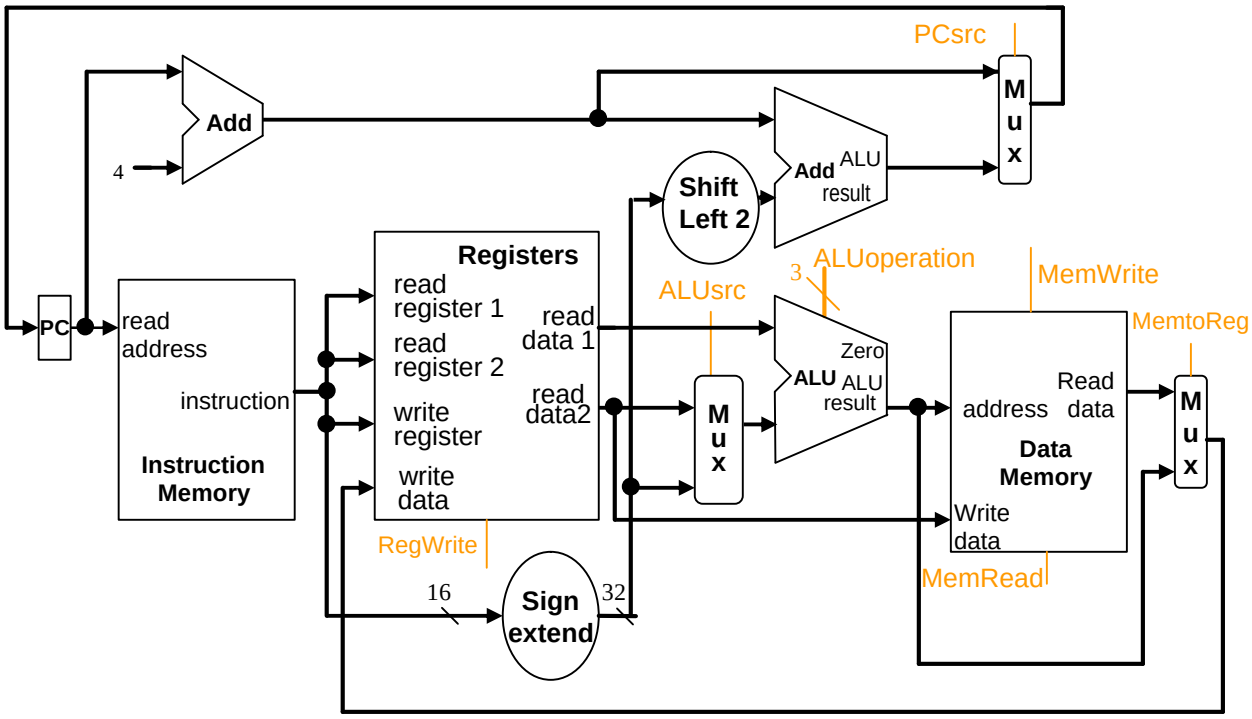


Data Memory Unit



Sign-Extension Unit

Building the Data-Path



Control #1

- Selecting the operations to perform (ALU, read, write, etc)
- Controling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of instruction
- Example:

instruction format

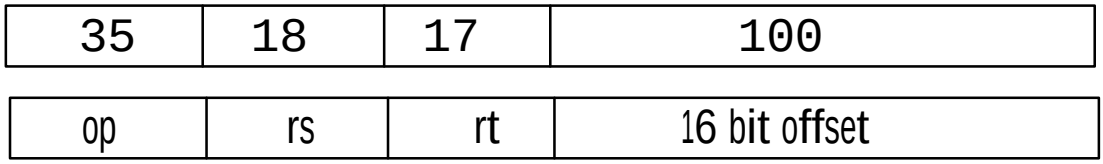
```
add $8,$17,$18
```

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

- ALU's operation based on instruction type and function

ALU Control #2

- What should the ALU do with this instruction
- **`lw $s1,100($s2)`**

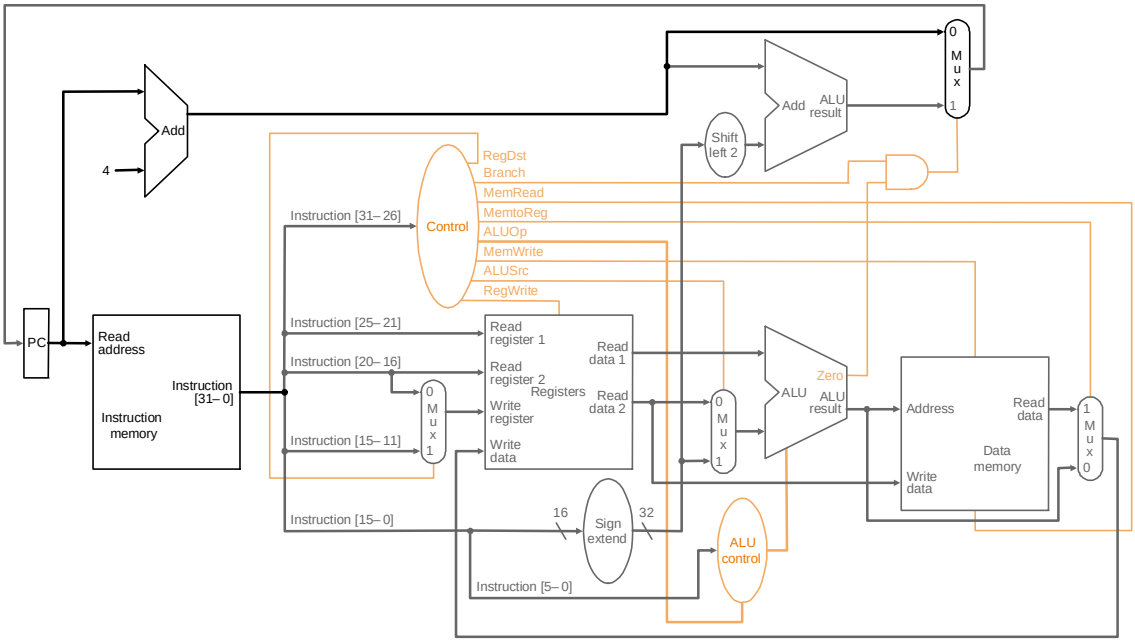


- Must describe hardware to compute ALU control input
  - given instruction type — desired ALU action
  - 00 = lw,sw (no funct field) — add
  - 01 = beq — subtract
  - 10 = arithmetic — add,sub,and,or,slt
  - function code for arithmetic

- ALU control input

000	AND
001	OR
010	add
110	subtract
111	set-on-less-than

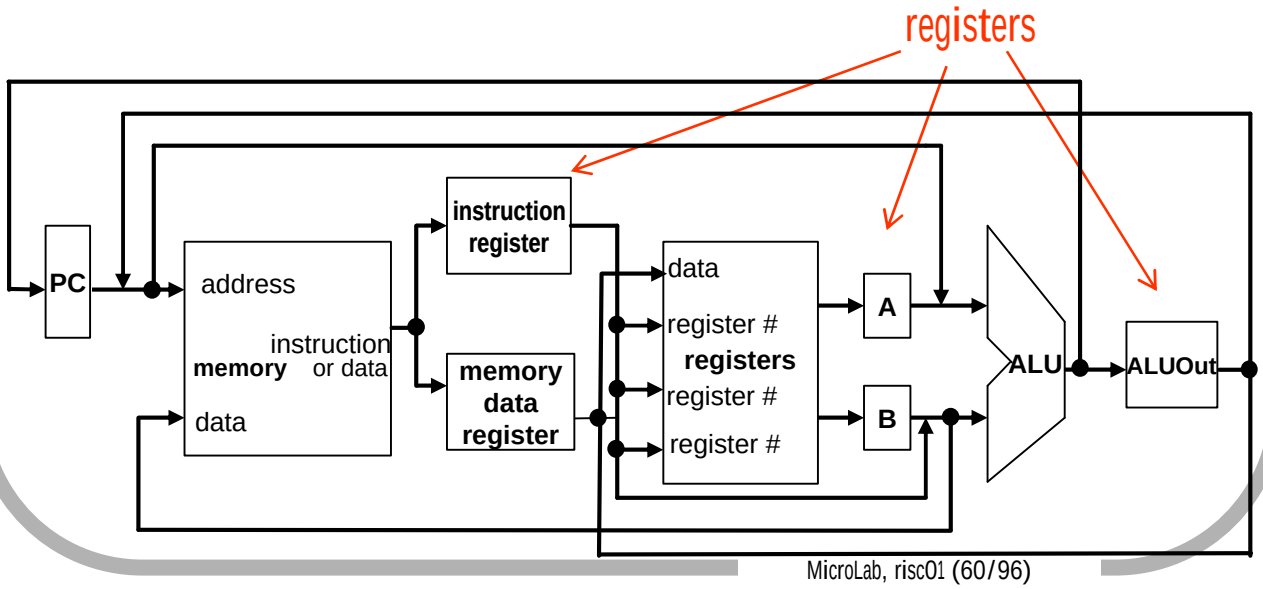
Flow Control #3



- Data-path operation(R-Type, **`add $s1,$s2,$s3`**)
  - fetch instruction from instruction memory, increment PC
  - read two registers from the register file ($s2,$s3)
  - ALU operates on register data using function code
  - result from ALU is written into destination reg $s1 of reg file
- Data-path operation(R-Type, **`lw $s1,off($s2)`**)
  - fetch instruction from instruction memory, increment PC
  - read register $s2 from the register file
  - ALU computes the sum $s2 and off
  - the sum from the ALU is used as address for the data memory
  - the data from memory unit is written into reg $s1 of register file

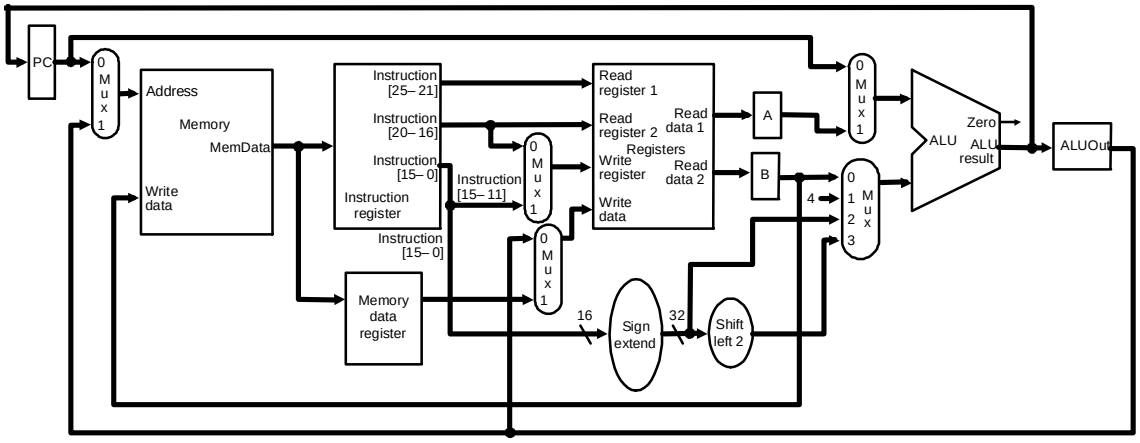
Single vs. Multiple Cycle

- Single cycle data-path
  - all of the logic is combinational
  - wait for everything to settle down and the right thing to be done
  - we use write signals along with clock to determine when to write
  - cycle time determined by length of longest path (compare R-Type instruction with lw instruction)
- Mulicycle cycle data-path
  - use smaler cycle time untis
  - have different instruction take different number of cycles
  - we will re-use functional units
    - ALU used to compute address and to increment PC
    - memory used for instruction and data
  - control signal will not be determined solely by instruction
  - we will use finit state machine for control



Multicycle Approach

- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use onyl one major functiuonal unit
- At the end of cycle
  - store values for use in later cycles
  - introduce additional „internal“ registers



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 to 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?

Step 2: Instruction Decode and Register Fetch

- Read register 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 +
(signextend(IR[15-0]) << 2;
```

sign-extended and shifted offset field

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

Step 3: (Instruction Dependent)

- ALU is performing one of the four functions,based on instruction type:

- Memory reference:
- **`ALUout = A + sigext(IR[15-0]);`**

- R-type:
- **`ALUout = A op B;`**

- Branch:
- **`if (A==B) then PC = ALUout;`**

- Jump:
- **`PC = PC[31-28]¦¦(IR[25-0]<<2);`**

Step 4: R-Type or Memory Access

- Load and store instruction access memory
- **`MDR = Memory[ALUout];`**
- or
- **`Memory[ALUout] = B;`**

- Arithmetic logical instruction (R-type):
- **`Reg[IR[15-11]] = ALUout;`**

- *the write actually takes place at the end of the cycle on the clock edge*

Step 5: Write Back

- During this last step, loads complete by writing back the value from memory

- **`Reg[IR[20-16]] = MDR;`**

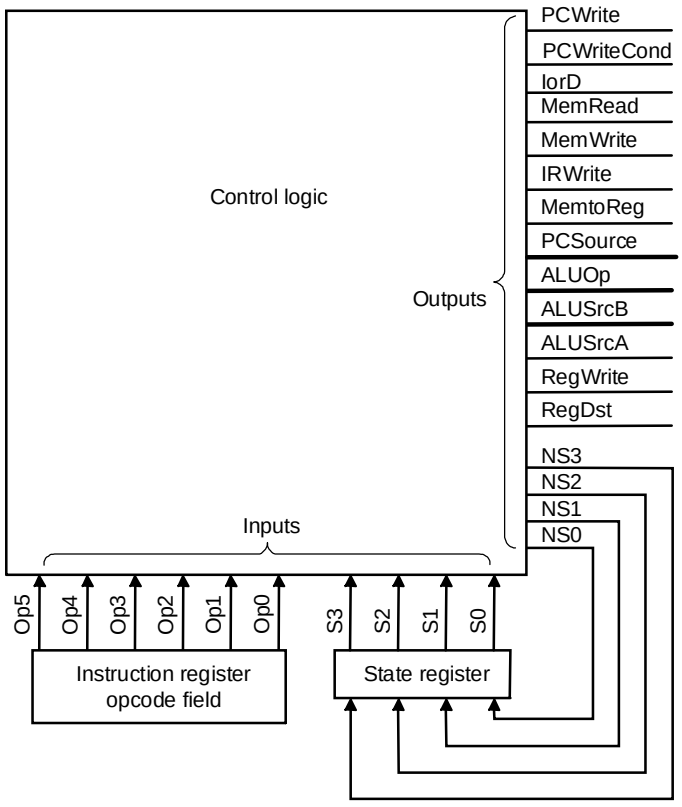
- Summary

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) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
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		

Implementing the Control

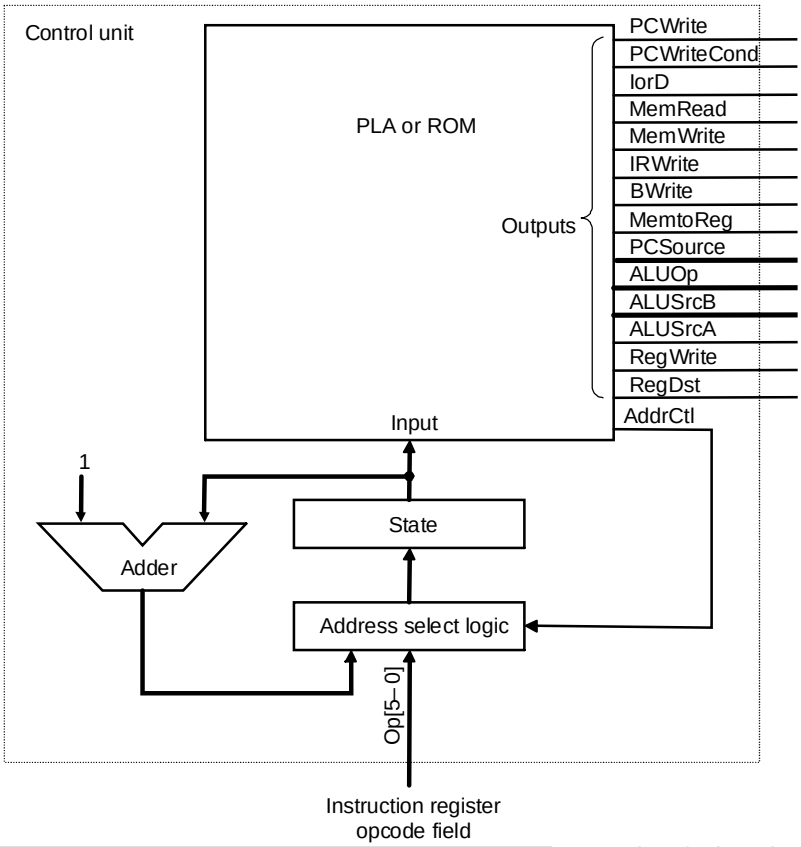
- Value of control signals depend on
  - what instruction is beeing executed
  - which step is beeing performed
- Implementation architectures
  - finit state machine: used for up to some dozen of states
  - microprogramming: used for hundreds or thousands of states

finit state machine



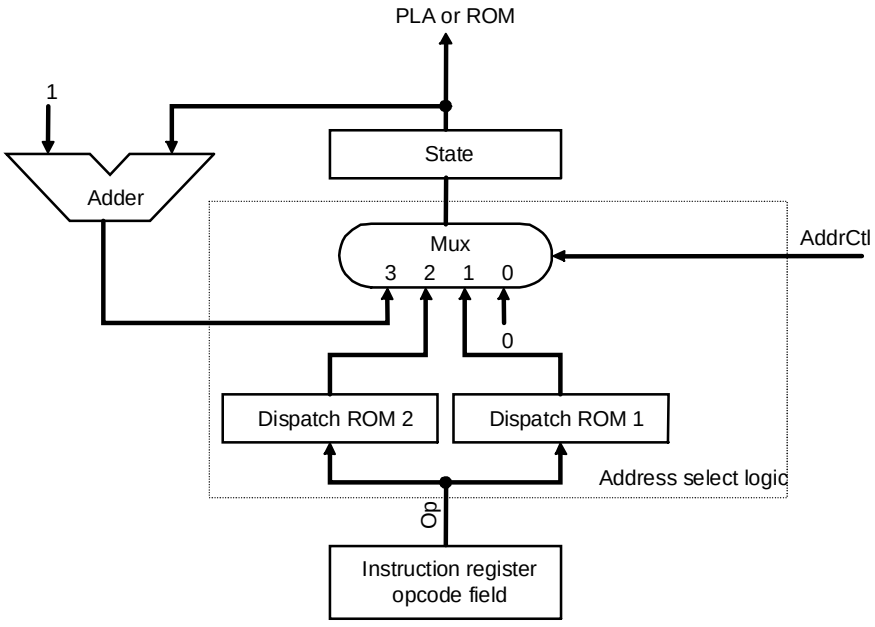
Control by Microprogramming

- Specification advantage
  - easy to design and write
  - design architecture and microcode in parallel
- Implementation advantages (off-chip ROM)
  - easy to change since values are in ROM
  - can emulate other architectures
  - can make use of internal registers
- Implementation disadvantages, slower now that:
  - control is implemented on same chip as processor
  - ROM is no longer faster than RAM



Microinstructions

- Microprogramming is based on microinstructions
  - each microinstruction defines the set of datapath control signals that must be asserted in a given state
  - a sequence of microinstructions defines one machine instruction
  - as microinstructions may execute sequentially, address increment logic is used (adder), for conditional branches and instruction dependent branches, dispatch ROMs are used
  - the key idea is to represent the asserted values on the control lines symbolically, so that the microprogram is a representation of the microinstructions, just as assembly language is a representation of the machine instructions



Microinstruction Sequencing

- Three different methods are available to choose the next microinstruction to be executed:
  - increment the address of the current microinstruction to obtain the address of the next microinstruction: Seq label
  - branch to microinstruction that begins execution of the next MIPS instruction: Fetch label
  - choose the next microinstruction based on the control unit input is called dispatch. Dispatch operations are implemented by creating a table containing the addresses of the target microinstructions. this table is indexed by the control unit input. We use 2 dispatch tables: Dispatch 1 label and Dispatch 2 label

Microinstruction Format

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, IorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, IorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, IorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

Designing Microinstructions

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

Label	ALU control	SRC1	SRC2	Register control	Memory	PCwrite control	Sequencing
fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1

Fields	Effect
ALU control, SRC1, SRC2	compute PC + 4 (is also written into ALUout though it will never be read from there)
Memory	fetch instruction into IR
PCwrite control	causes output of the ALU to be written into the PC
Sequencing	Go to the next microinstruction

Fields	Effect
ALU control, SRC1, SRC2	store PC + signextens IR[15-0] < < 2 into ALUout
Register control	use rs and rt fields to read reg placing data in A,B
Sequencing	use dispatch table 1 to choose the next micro addr

Designing Microinstructions

Label	ALU control	SRC1	SRC2	Register control	Memory	PCwrite control	Sequencing
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch 1

Fields	Effect
ALU control, SRC1, SRC2	ALU operates on the contents of the A and B reg using function field to specify the ALU operation
Sequencing	Go to the next microinstruction

Fields	Effect
register control	the value in ALUout is written into the register file entry specified by the rd field
Sequencing	go to the next micro instruction labeld fetch

Microprogramming for our MIPS

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

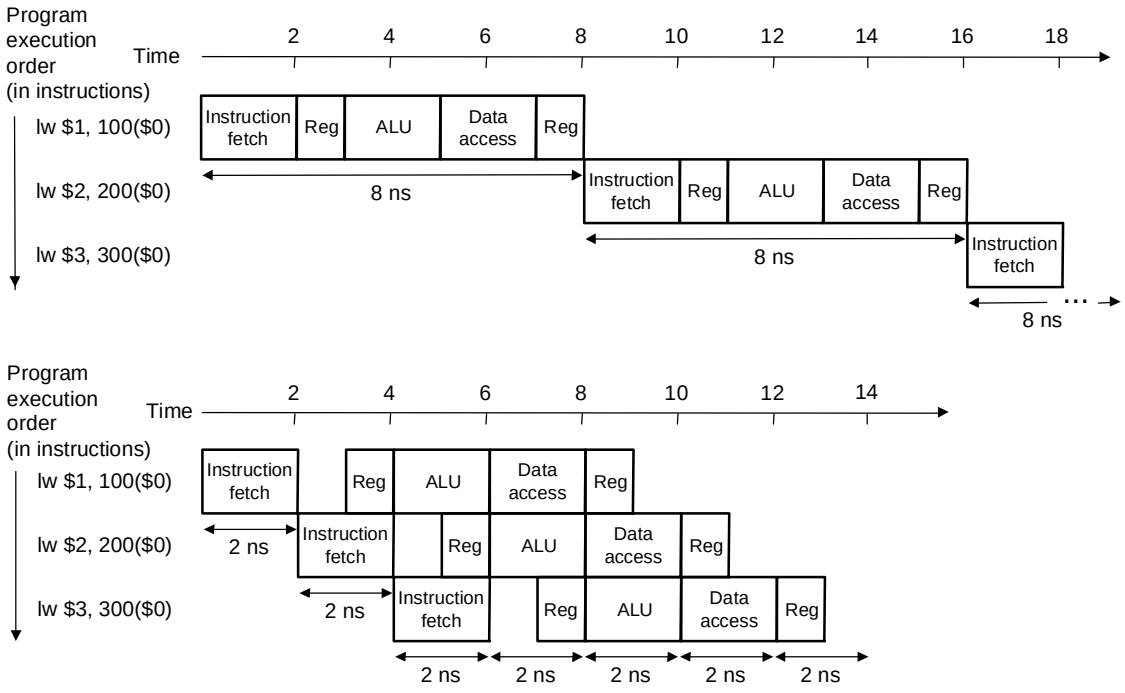
Exceptions & Interrupts

- one of the hardest part of control is implementing exceptions and interrupts
- they change the normal flow of instruction execution
- an interrupt is an event that causes unexpected change of control flow but comes from outside of the processor (I/O device request)
- an exception is an unexpected event from within the processor (arithmetic overflow, undefined instruction, invoke OS from user program)

- handling exceptions:
  - address of offending instruction saved in exception program counter (EPC)
  - transfer control to OS at some specified address
  - cause register holds status information with reason for exception

Pipelining

- pipelining is an implementation technique in which multiple instructions are overlapped in execution
- under ideal conditions, the speedup from pipelining equals the number of pipe stages
- pipelining increases instruction throughput
- pipelining increases latency



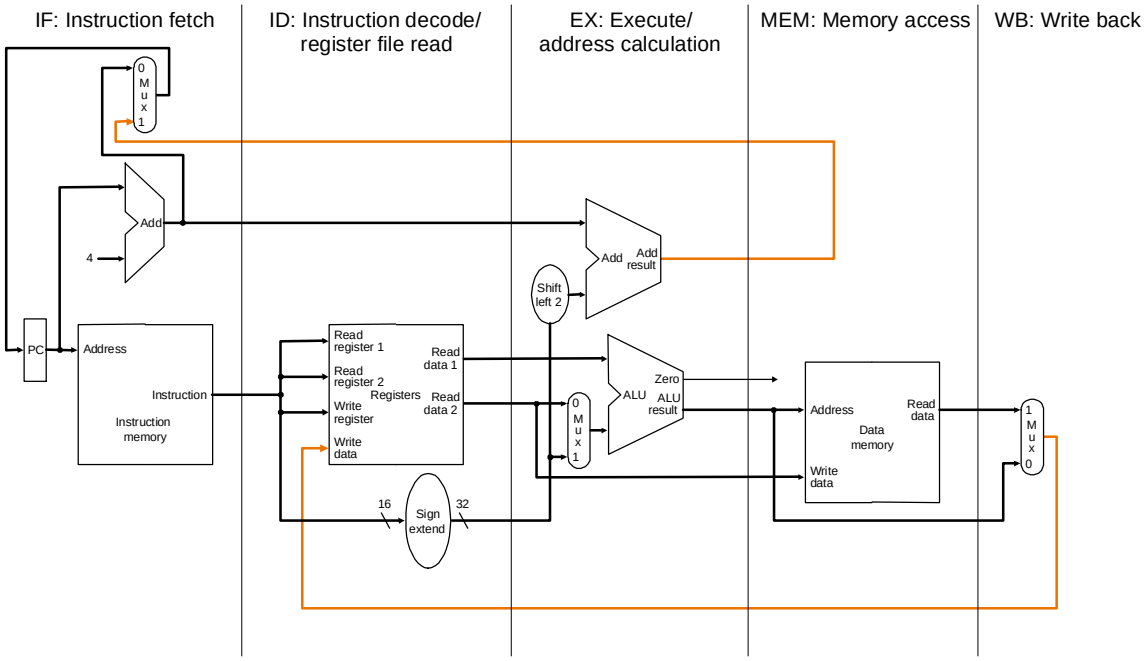
Pipelining: Hazards

- what makes it easy
  - all instruction are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
- definition of hazard: hardware cannot support combination of instructions that we want to execute in the same clock cycle
- what makes it hard
  - structural hazards: suppose we have only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on previous instructions
- we'll built a simple pipeline and look at these issues
- we'll talk about modern processors and what really makes it hard:
  - exception handling
  - trying to improve performance with out-of-order execution, etc.

Basic Idea

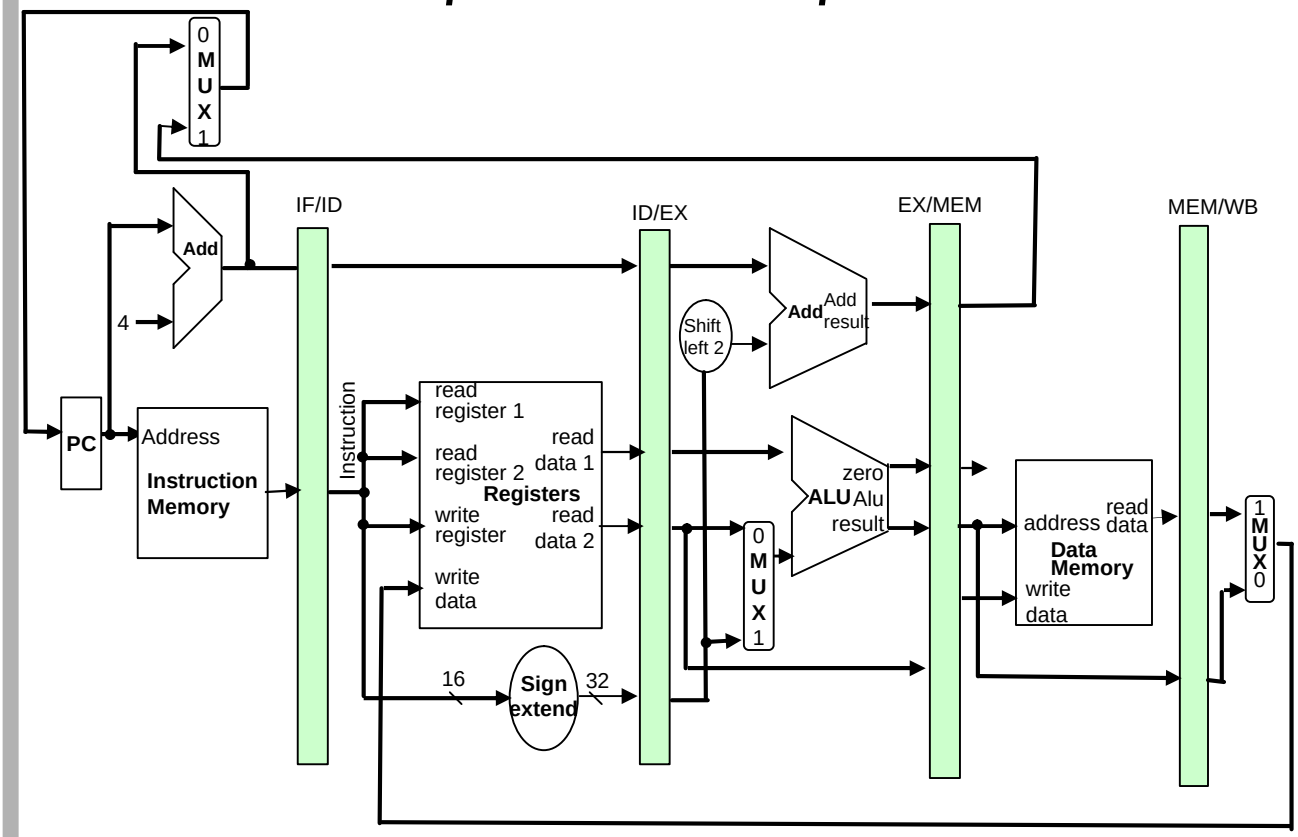
five stage pipeline

- IF: instruction fetch
- ID: instruction decode and register file read
- EX: execution or address calculation
- MEM: data memory access
- WB: write back

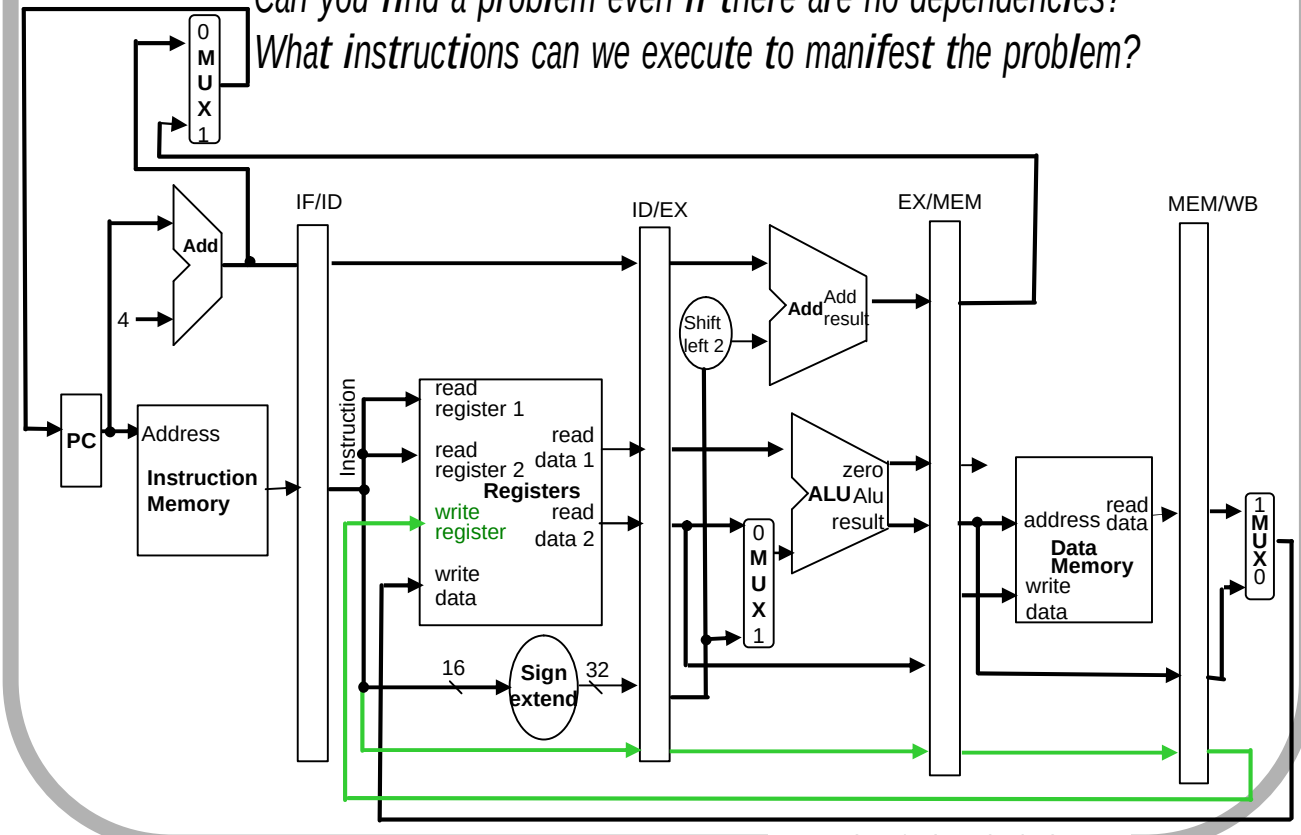


What do we need to add to actually split the datapath into stages?

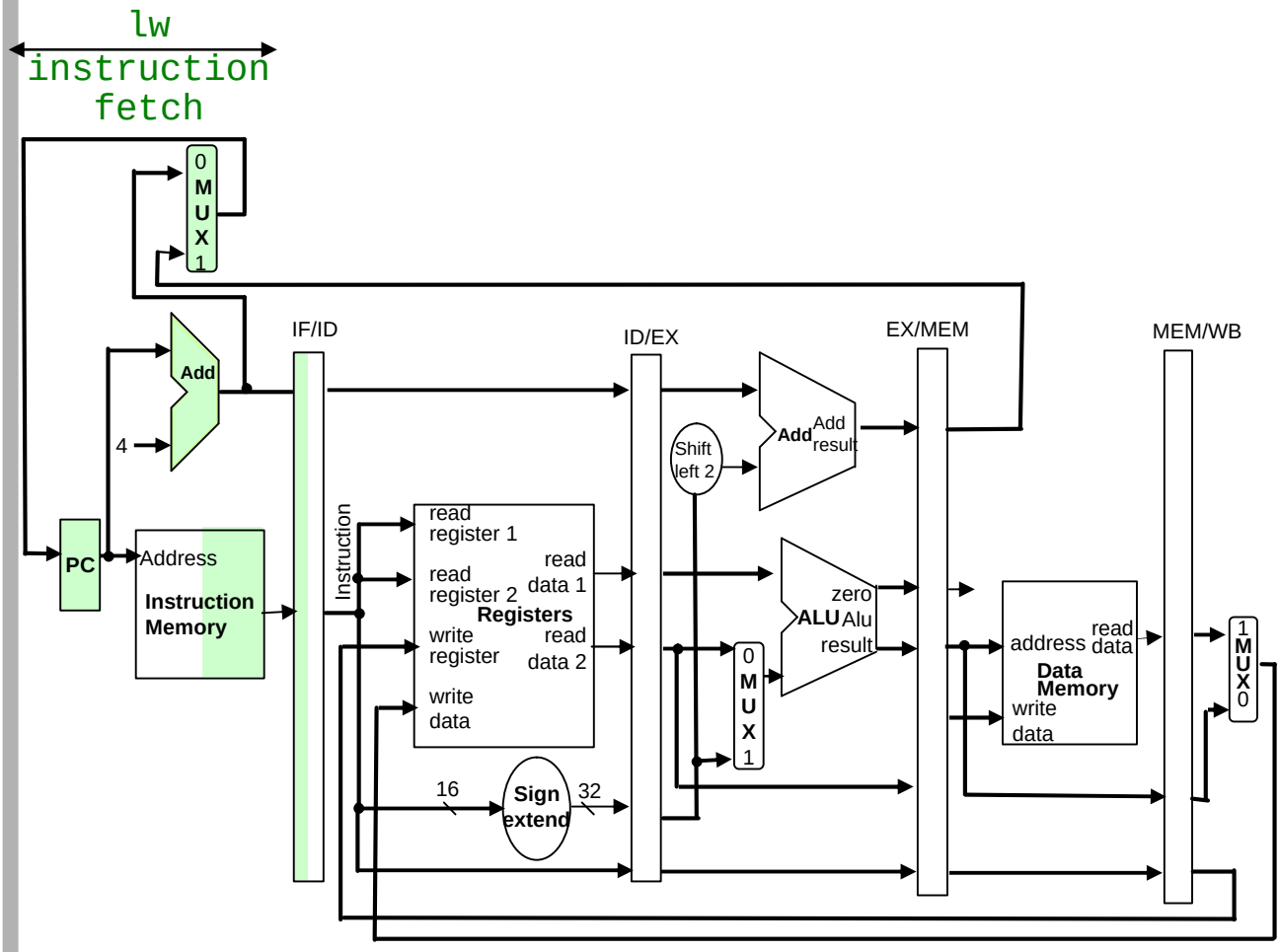
Pipelined Datapath



Can you find a problem even if there are no dependencies?
What instructions can we execute to manifest the problem?

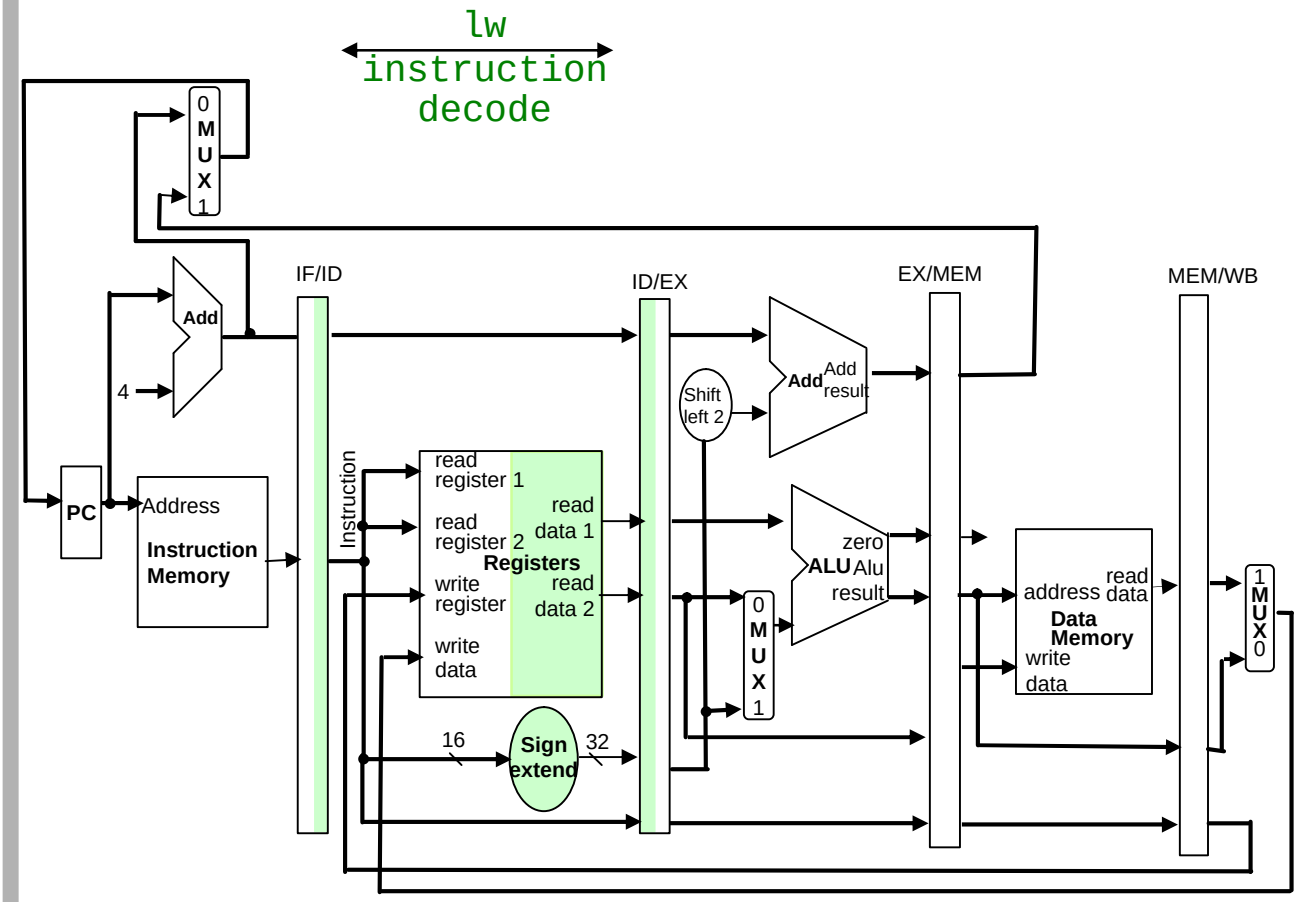


Instruction Fetch (Working Pipeline: Step 1)



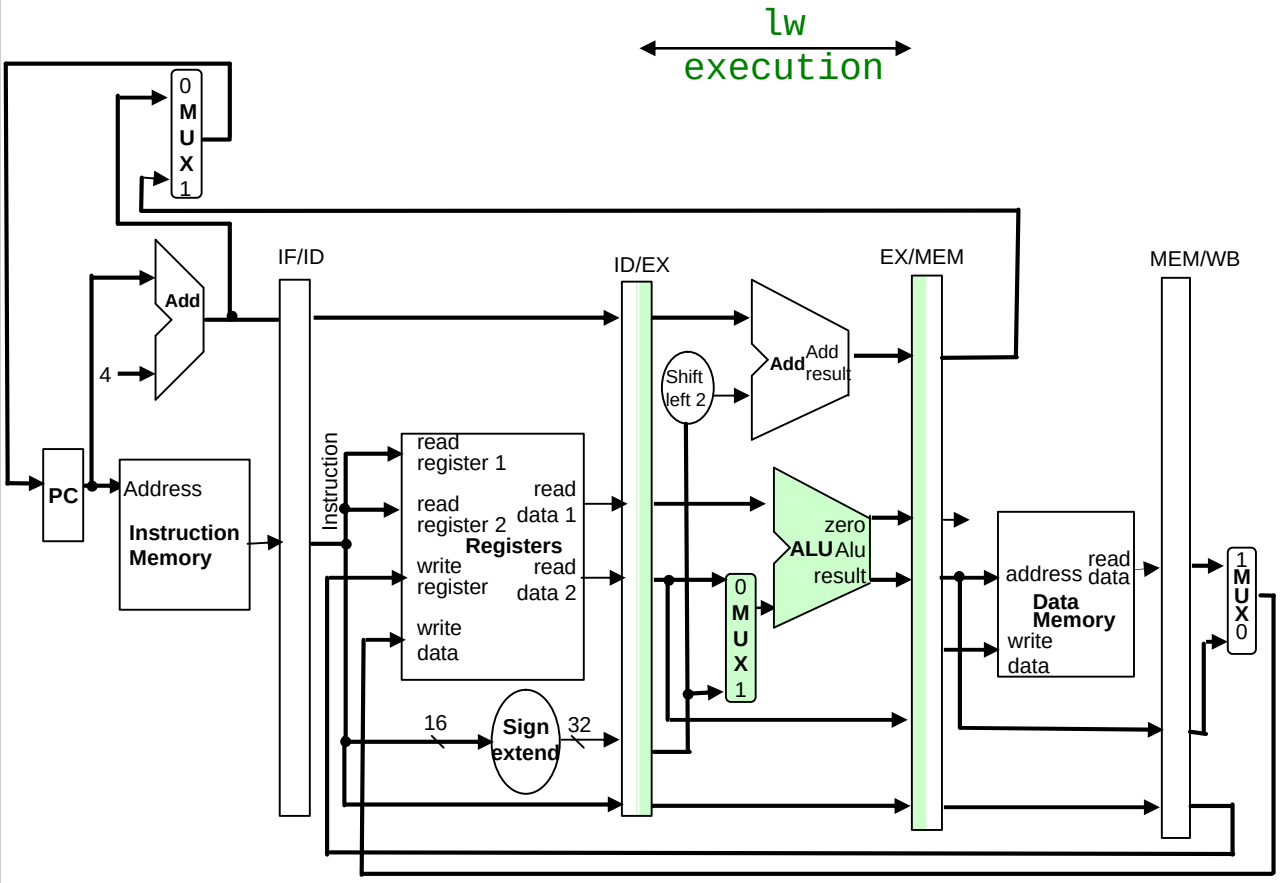
The instruction is beeing read from the memory using the address in the PC and then placed in the IF/ID pipeline register. The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle. This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as beq. The computer cannot know which type of instruction is beeing fetched, so it must prepare for any instruction, passing potentially needed information down to the pipeline.

Instruction Decode and Reg File Read (Working Pipeline: Step 2)



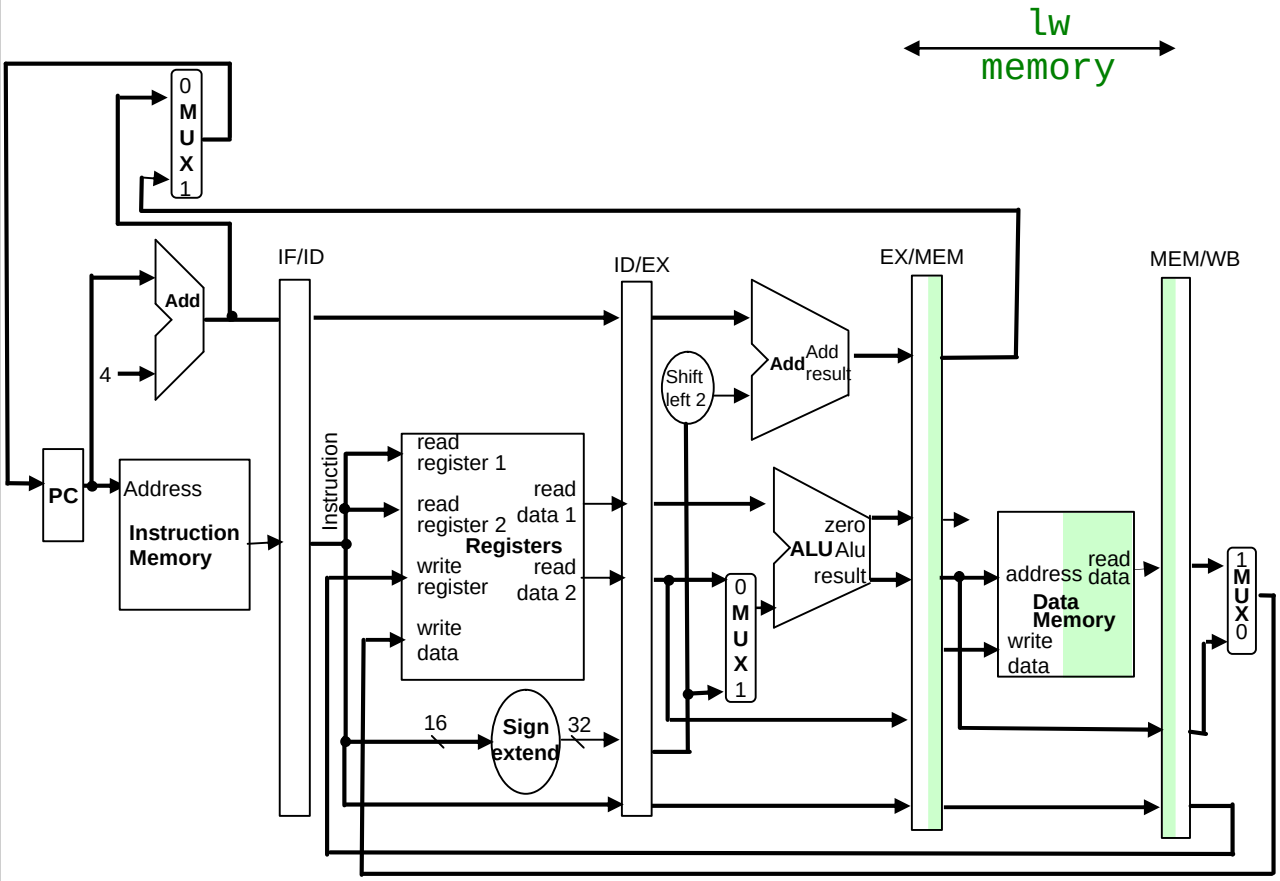
The instruction portion of the IF/ID pipeline register supplies the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers. All three values are stored in the ID/EX pipeline register, along with the incremented PC address. We again transfer everything that might be needed by any instruction during a later clock cycle.

Execute or Address Calculation (Working Pipeline: Step 3)



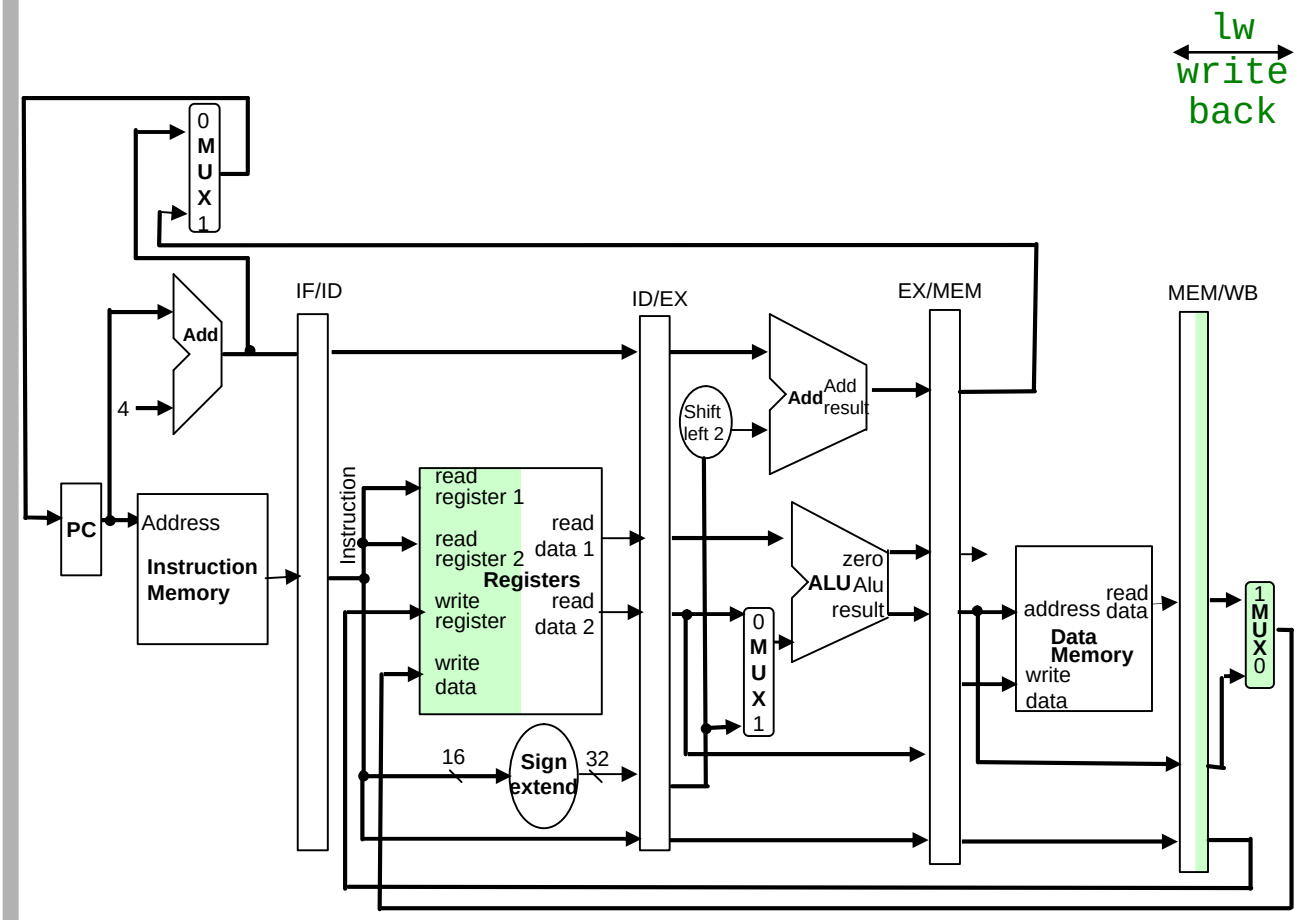
The load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU. The sum is placed in the EX/MEM pipeline register.

Memory Access (Working Pipeline: Step 4)



The load instruction is reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.

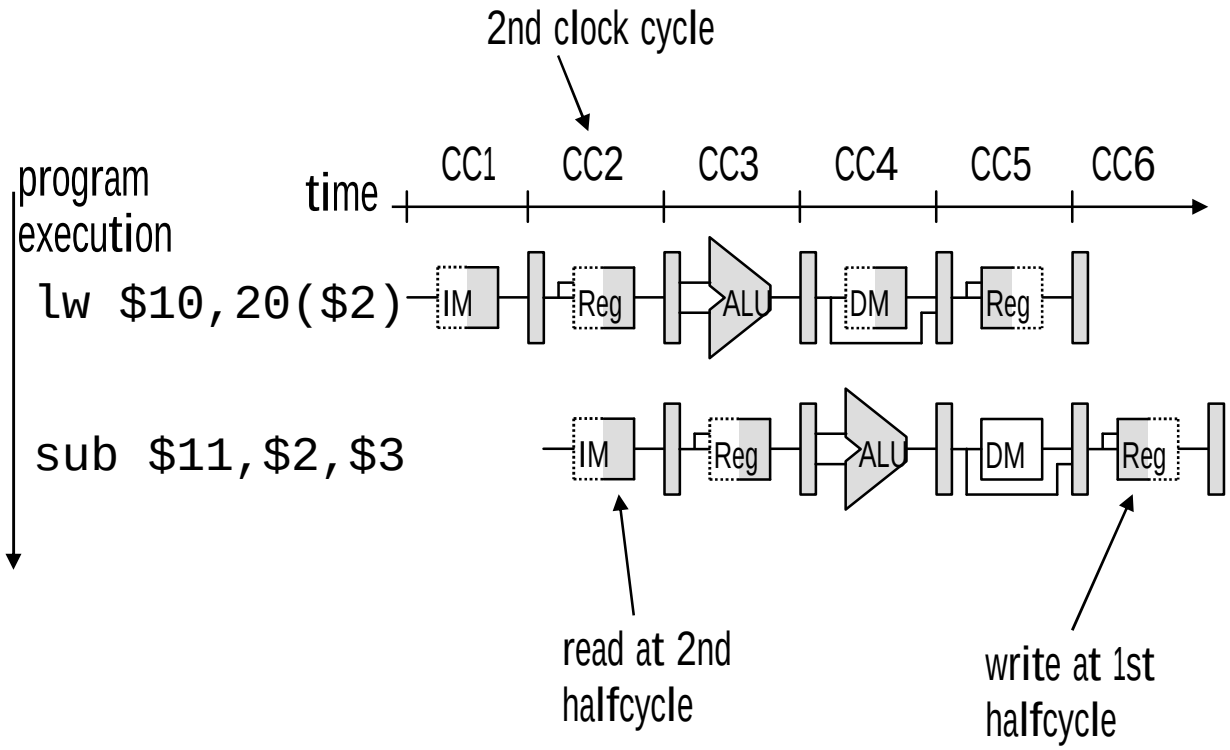
Write Back
(Working Pipeline: Step 5)



In the final step, the data is being read from the MEM/WB pipeline register and written back into the register file.

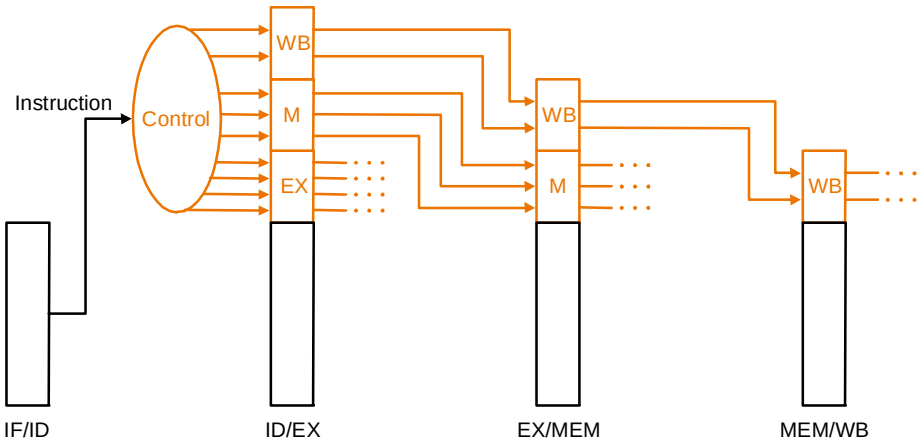
Graphical Representation

- IM: instruction memory
- Reg: register file
- ALU
- DM: data memory

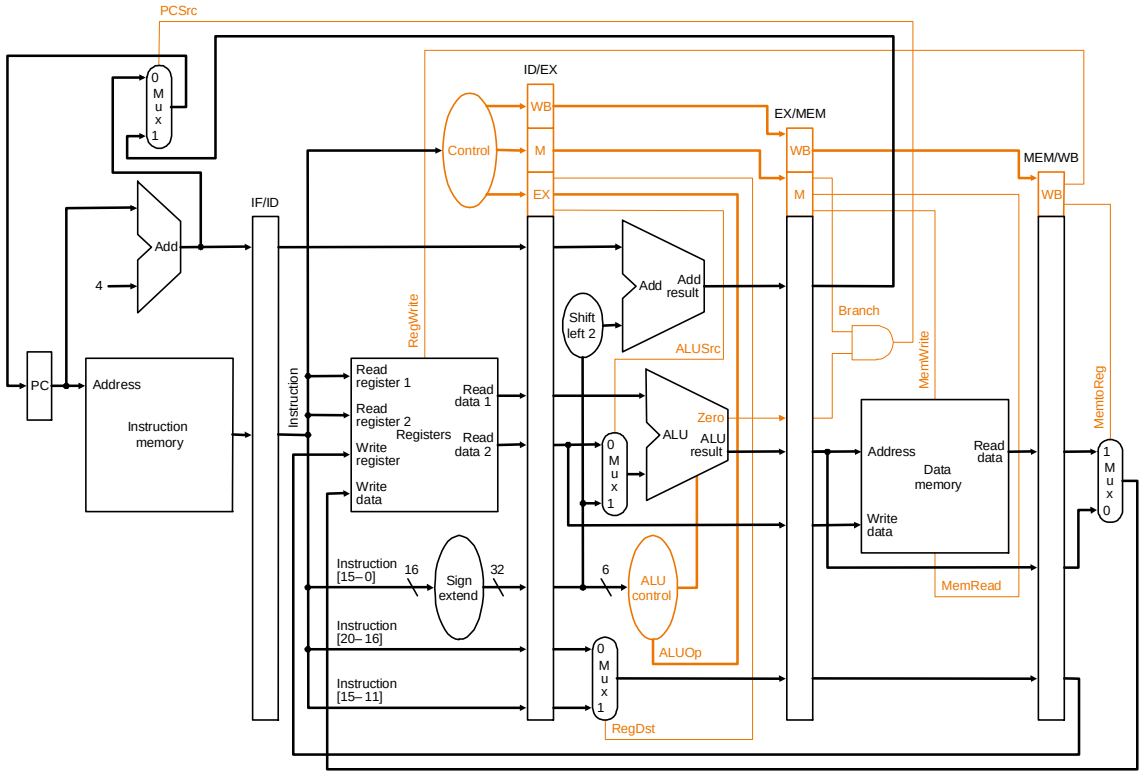


Pipeline Control

- we have 5 stages, what needs to be controled in each stage?
  - instruction fetch, PC increment
  - instruction decode, register fetch
  - execution
  - memory stage
  - write back
- pass control signals along just like the data

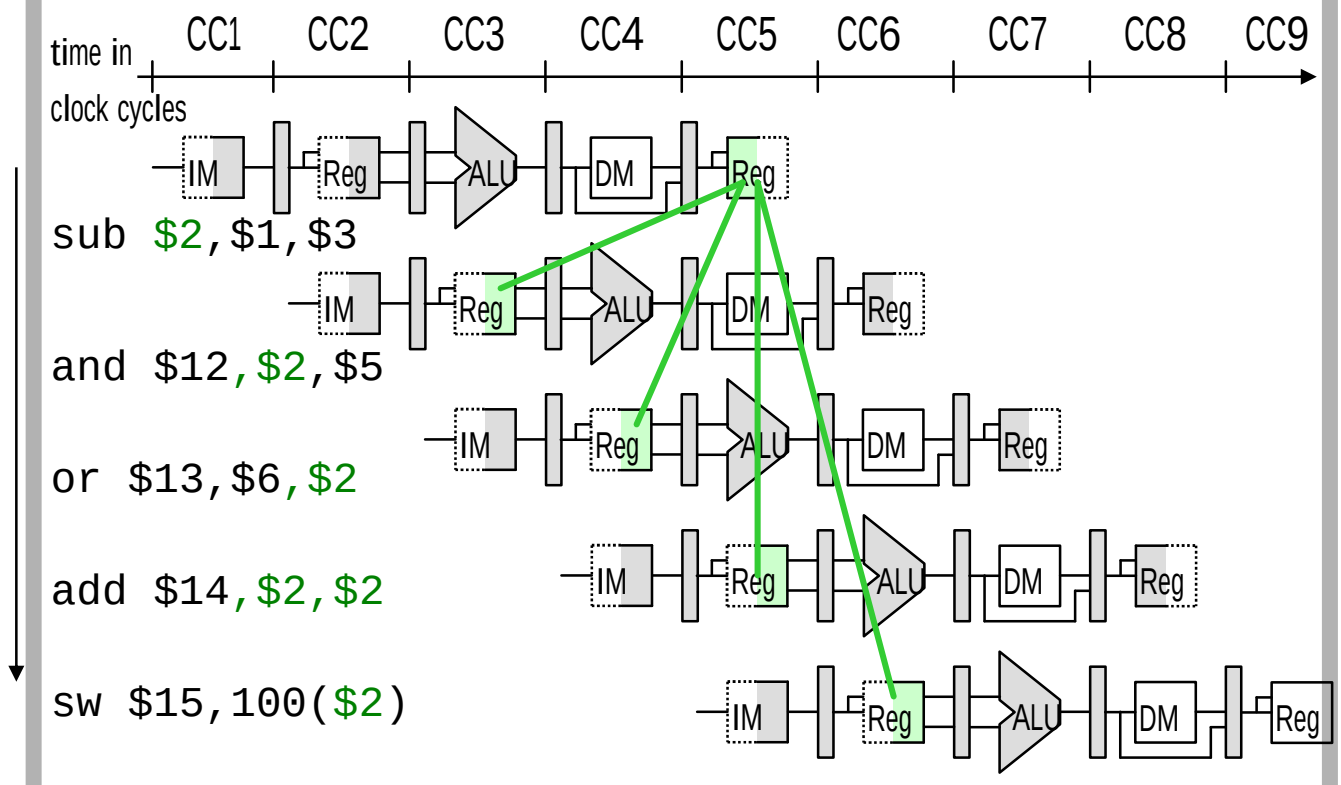


Datapath with Control



Data Hazards #1

Problem with starting next instruction before first is finished
- dependencies that go backward in time are „data hazards“



Data Hazards #2

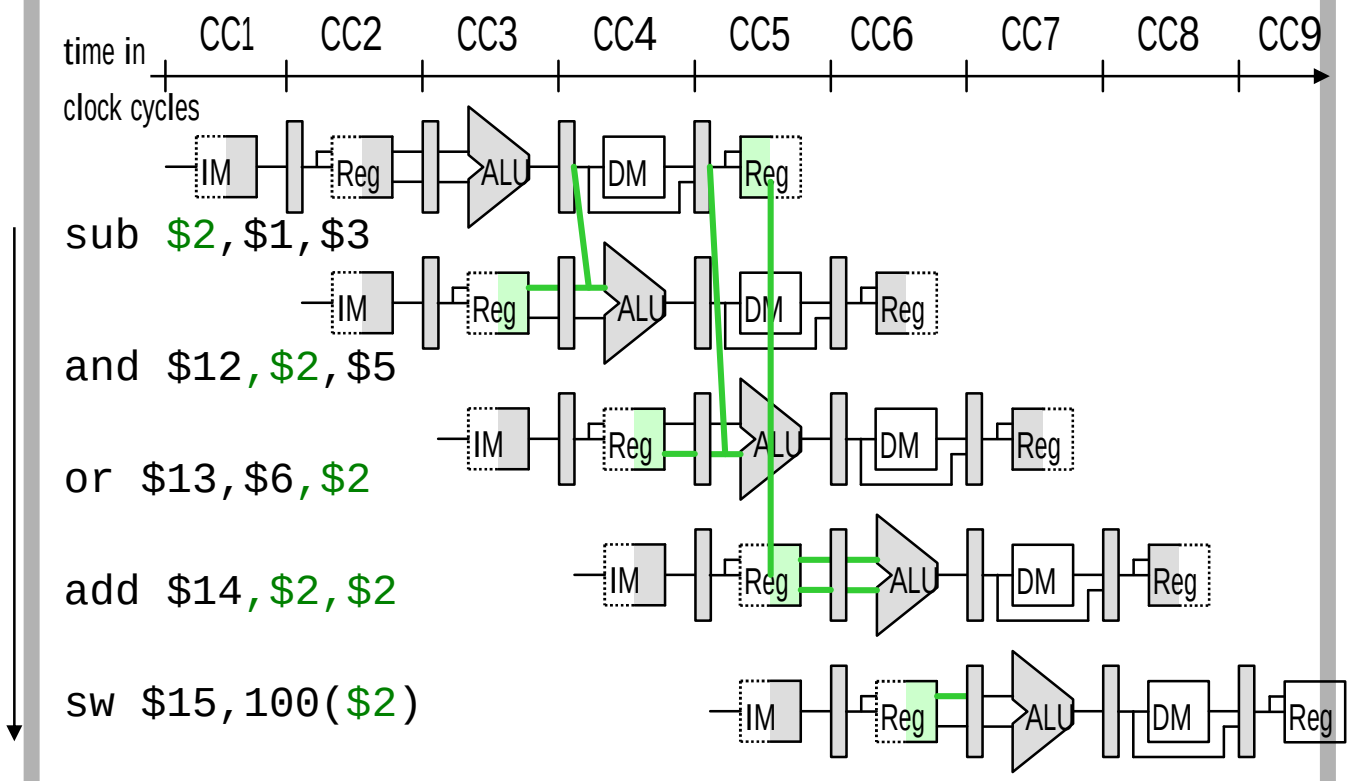
- software solution
  - have a compiler guarantee no hazards
  - where do we insert the „nops“

```
sub     $s2,$s1,$s3
and     $s12,$s2,$s5
or      $s13,$s6,$s2
add     $s14,$s2,$s2
sw      $s15,100($s2)
```

  - problem: this really slows us down

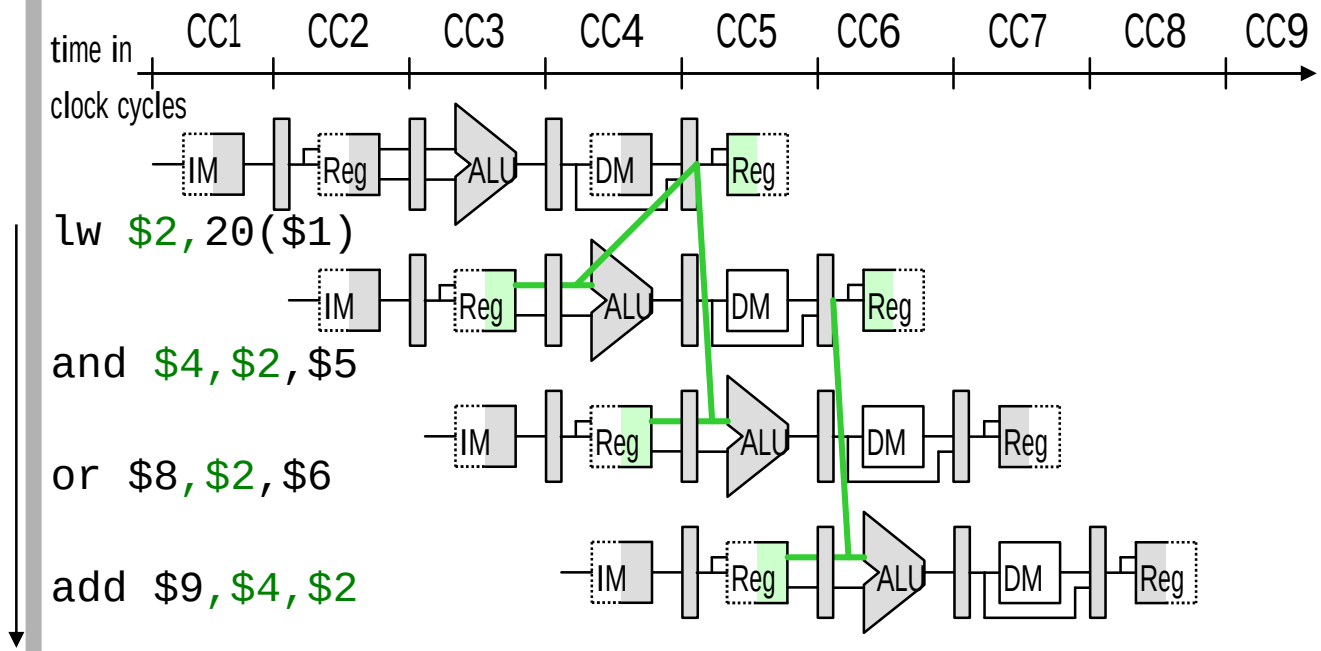
Data Hazards: Forwarding

- the dependencies between the pipeline registers move forward in time



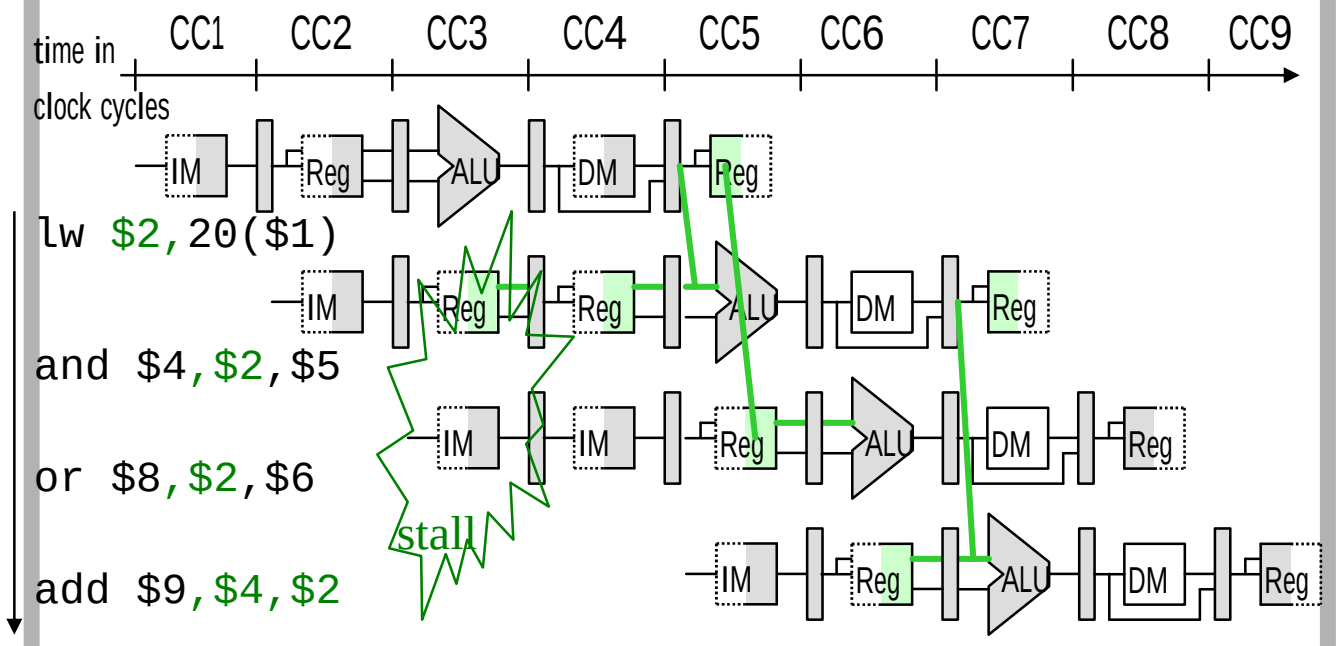
Data Hazards: Stalls

- since the dependence between the **lw** and the following **and** instruction goes backward in time, this hazard cannot be solved by forwarding
- this combination must result in a stall by the hazard detection unit



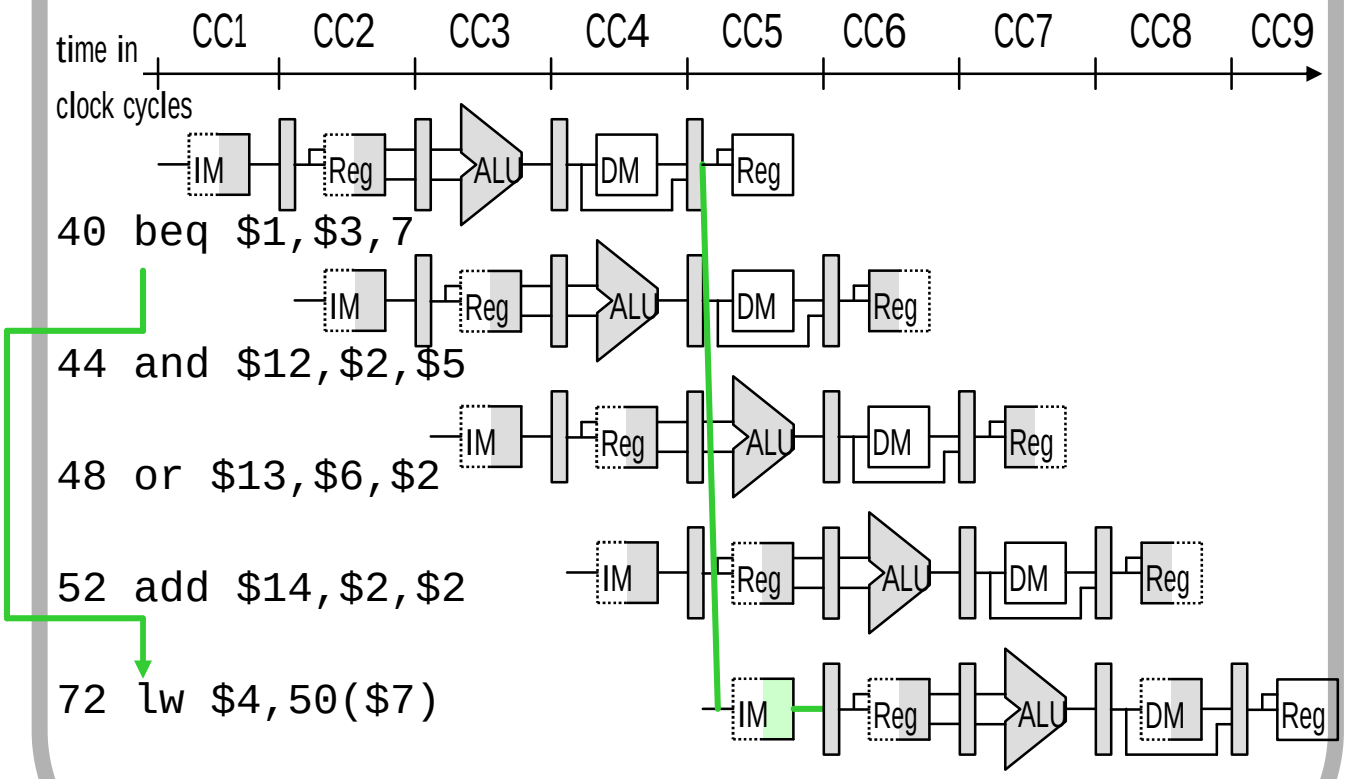
Data Hazards: Stall Insertion

- we can stall the pipeline by keeping an instruction in the same stage



Branch Hazards

- the branch instruction takes its decision in the DM stage
- when we decide to branch, other instructions would already be in the pipeline
- we are predicting „branch not taken“ (need to add hardware for flushing instructions if we are wrong)



Improving Performance

- Try to avoid stalls
  - eg: reorder these instruction

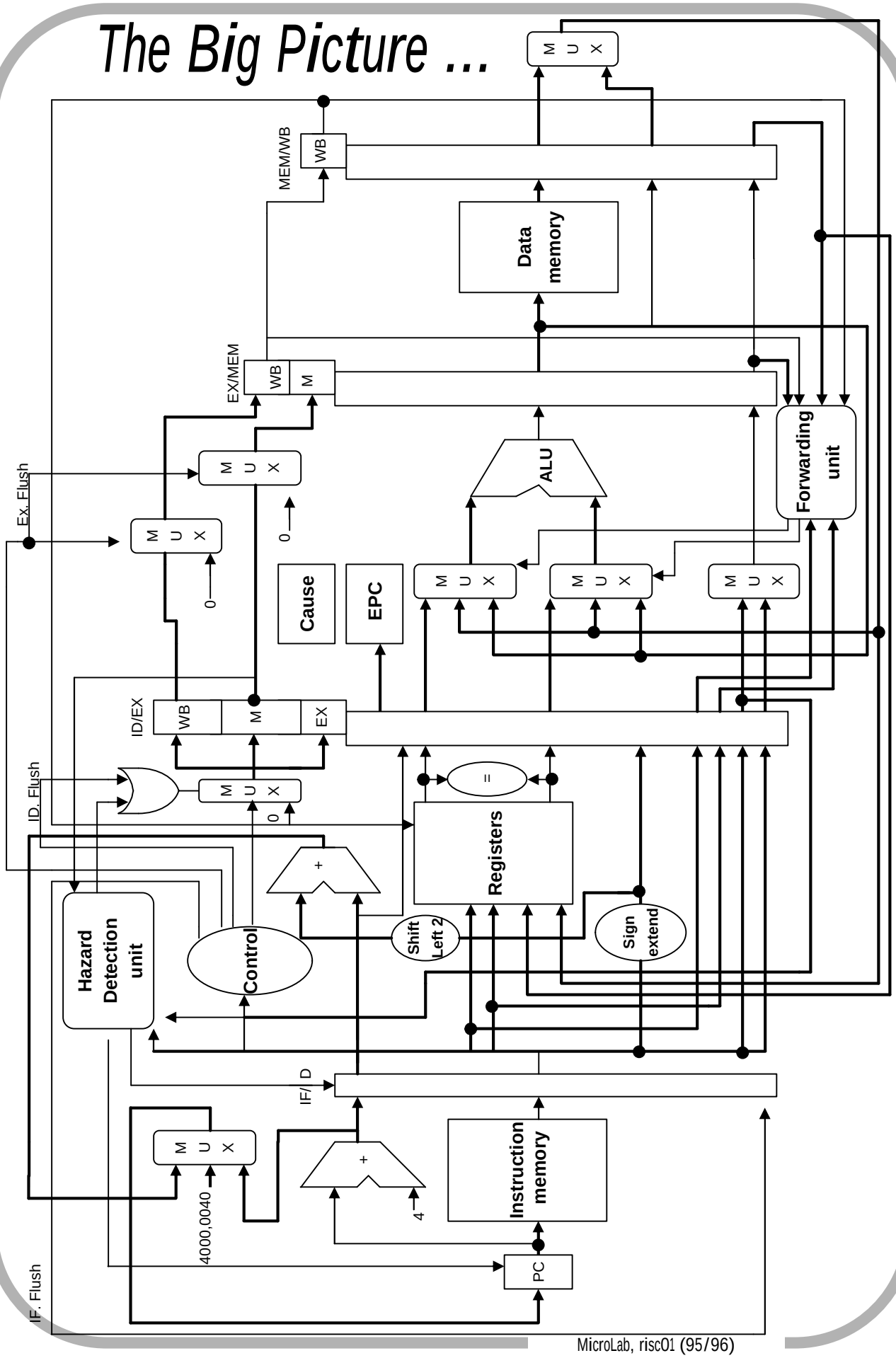
```
lw t$0,0($t1)
lw t$2,4($t1)
sw $t2,0($t1)
sw $t0,4($t1)
```

- Add a branch delay slot
  - the next instruction after a branch is alwyas executed
  - rely on compiler to fill the slot with something useful

- dynamic scheduling
  - the hardware performs the scheduling
    - hardware tries to find instructions to execute
    - out of order execution is possible
    - speculative execution and dynamic branch prediction (branch prediction buffer or branch history table)

- superscaler: start more than one instruction in the same cycle

The Big Picture ...



MicroLab, risc01 (95/96)

To Probe Further ...

- The book from Hennessy&Patterson addresses much more important topis:
  - caches
  - virtual memory
  - memory hierarchy
  - design of I/O systems
  - buses
  - multiprocessors
  - clusters
  - network topologies
  - ...

- There is a second book from the same authors adressing the same subject: ISBN 1-55860-596-7 (May 2002)