CS 201

Digital Logic

Intro to Assembly Language

Britton MIPS text

CPU design

- We have everything we need
- None of it does anything without our direct manipulation
- Humans are slow, and single operation computers are useless
- ***Automate*** the behaviour we have devised and are seeing in the lab
- You are learning the software side in the lab - we'll get there, so don't worry if the class and the lab aren't lined up quite yet...

Fetch-Execute cycle

- CPU uses program instructions to direct operation.
- Instructions are binary strings stored in memory
- Operation proceeds in a fetch-execute cycle
  - ***Fetch*** an instruction from memory
  - ***Execute*** the instruction
    - Decide where to fetch the *next* instruction

Instructions

- High level language is translated into machine language: 1s and 0s
- Hard to read or write by human.
- Assembly language: an intermediate step
  - Direct translation up from binary
  - (almost) human readable
- CPU design: hardware that will fetch and execute these instructions

Instruction format

- Basic idea:

Opcode	Operand

  - Opcode = what to do, Operand = what to do it to
- 3-operand

  op3 = op1 + op2

opcode	operand 1	operand 2	operand 3

- 2-operand

  op2 = op1 + op2

opcode	operand 1	operand 2

- 1-operand

  Register = Register + op1

opcode	operand 1

- 0-address

  stack(0) = stack(1)+stack(0)

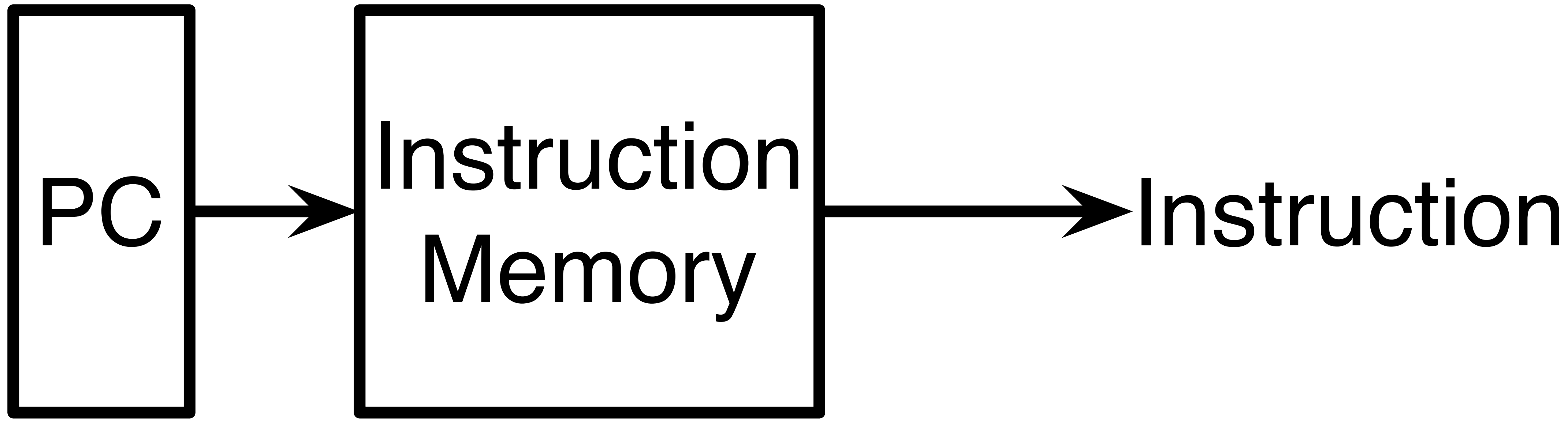
opcode

CPU Hardware: Incremental design

  - Note: this will be a generic computer.
  - Once we get the feel, we'll add some specific instructions
- Fetch hardware:
  - Memory to get the instruction from
  - Address to indicate memory location of instruction
    - called ***Program Counter*** (PC)

Fetch Hardware

- To fetch: need a program counter and some memory:
- the PC addresses memory
  - address from PC goes to input of decoder of instruction memory
  - Instruction from memory is output



Fetch Hardware

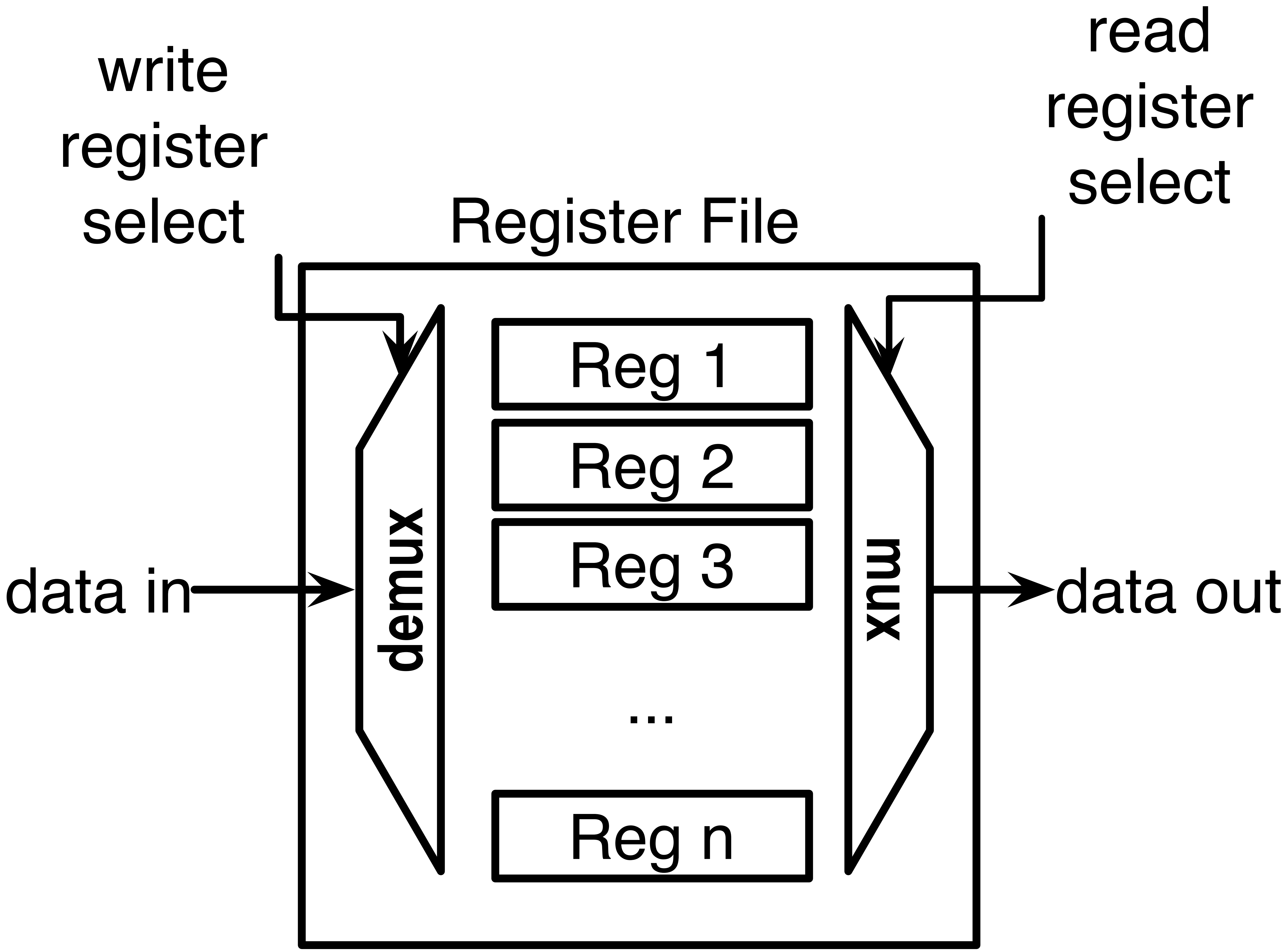
- PC Register currently only needs to hold a value
  - D flip flops are enough (more later)
- Memory currently only holds instruction data
  - can be ROM, as shown earlier
  - RAM is also possible, as shown earlier.
- PC provides ***address*** of the instruction to be fetched
  - address must be decoded into the memory word line
  - 32 bit PC means 4 billion possible memory words
  - in practice, this is heavily constrained

Hardware to do math

- Basic function of computer (to do math)
- recall ALU which we designed last section
- also need registers to hold input and output values
  - Can have a bunch of special purpose registers
  - In our model, we will use a ***register file***
    - a collection of registers that can be read from or written to
    - a decoder choses which register is selected
    - very much like memory

Basic register file

- Demultiplexer selects a register to write to
- Multiplexer selects a register to read from
- Can have multiple read multiplexers to fetch more than one operand
- **[rs]** is notation for contents of register indicated by selection *rs*



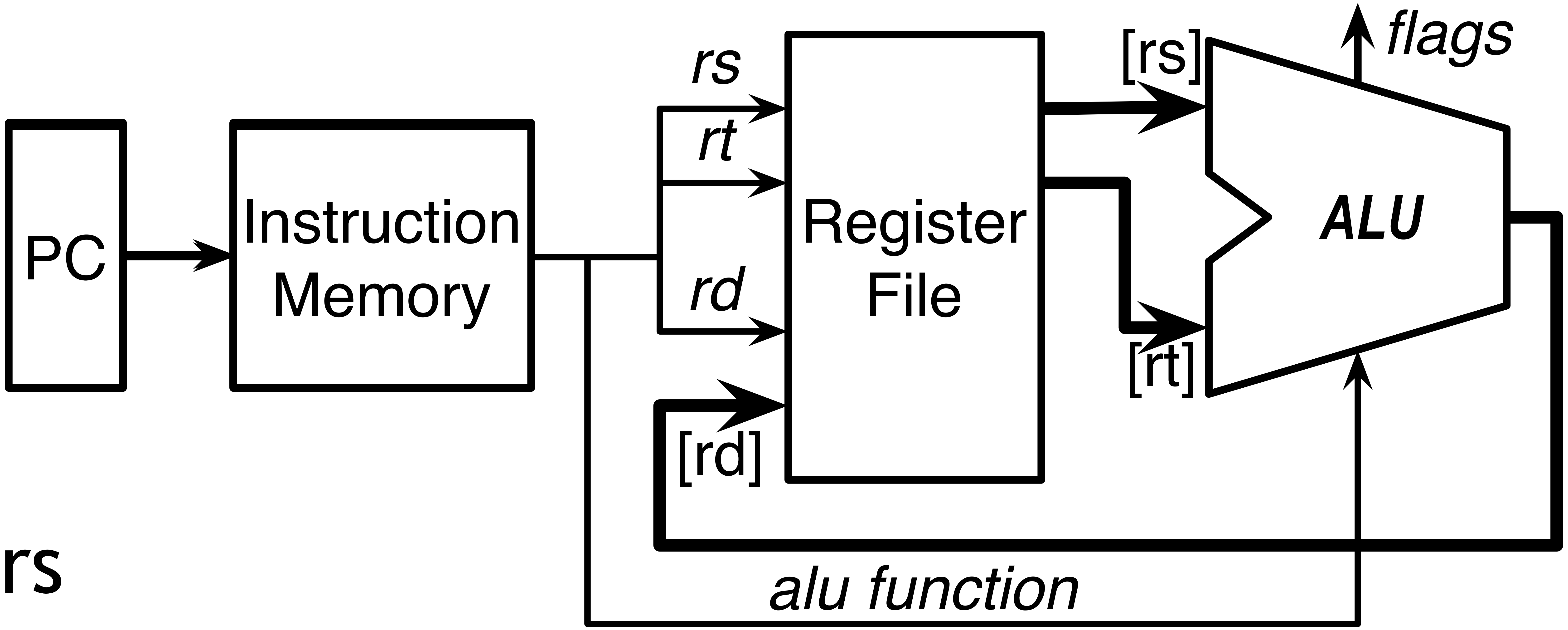
Register File for our generic computer

- We'll build a machine with 3 operands
  - 2 source operands, one destination operand
  - so we need two data out multiplexers, and one data-in demultiplexer
- The instruction must contain the ***address*** for each register to be used
  - e.g.: add register 3 and register 4 together, and put the result in register 6
    - value(0110) = value(0011) + value(0100)

Adding a Register File and ALU

label = register address

[label] = data from or to register



To add two numbers

1. PC addresses Instruction Memory
2. Instruction indicates which registers to use
3. Source registers (*rs*, *rt*) are extracted from register file [rs] and [rt] and routed to ALU
4. ALU result [rd] is moved to destination register (*rd*)

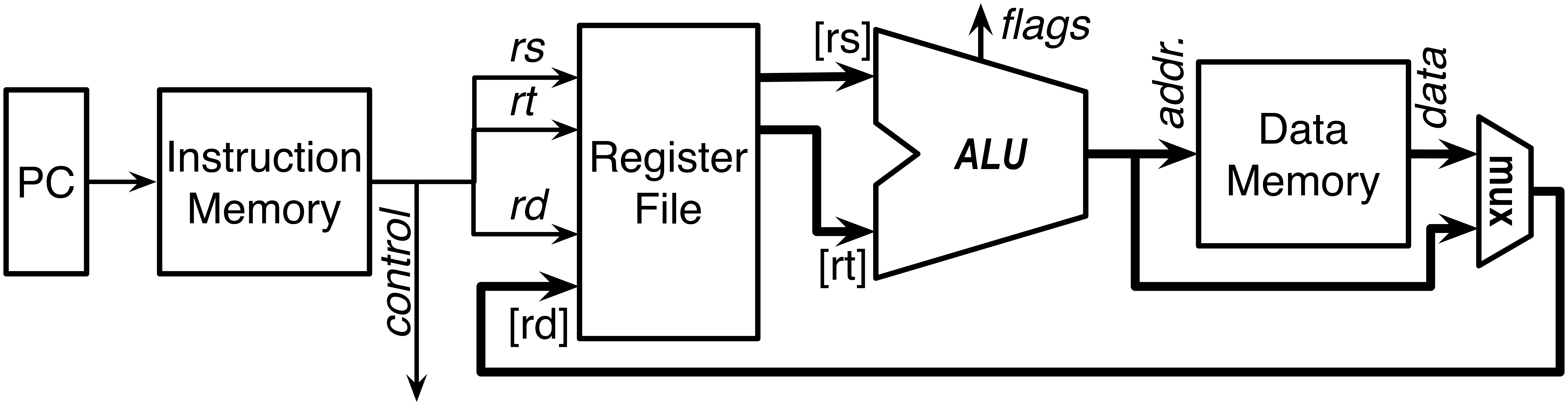
More Memory

- Summing the contents of registers is great…
  - where does that data come from?
  - what if we need more storage?
- We need some data memory
  - store and retrieve data from/to registers
  - hold big chunks of data that won’t fit in our register file all at once
- Why not just make an enormous register file?
  - Registers are very fast but very expensive, so small.
  - Main memory can be cheaper, and so bigger

Adding Data memory

- Need to be able to move data between register file and memory
- Need to be able to address the memory
  - where to put the result?
- Memory Address will come from the instruction as well
  - Lots of ways to address data, as we'll see
  - for flexibility, we will use the output of the ALU as the source of our address
    - can calculate complex addresses if needed.

Adding Data memory

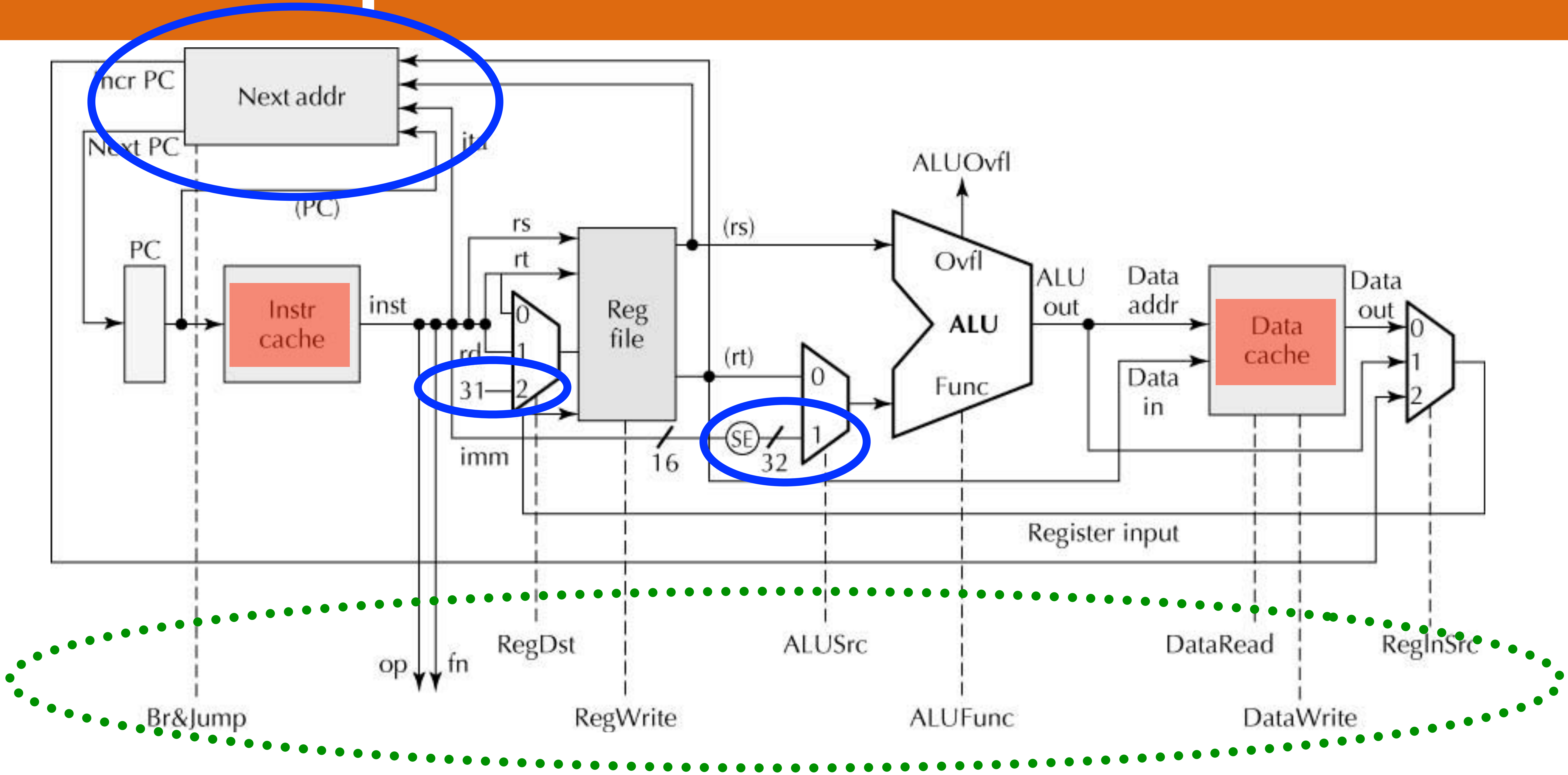


- ALU output now can be used as an address for data memory, or destination register value.
- Reg file write could come from ALU or Data Memory, so we must choose which one using a multiplexer
- generic "control" from instruction mea for ALU, mux, mem etc

MIPS architecture target

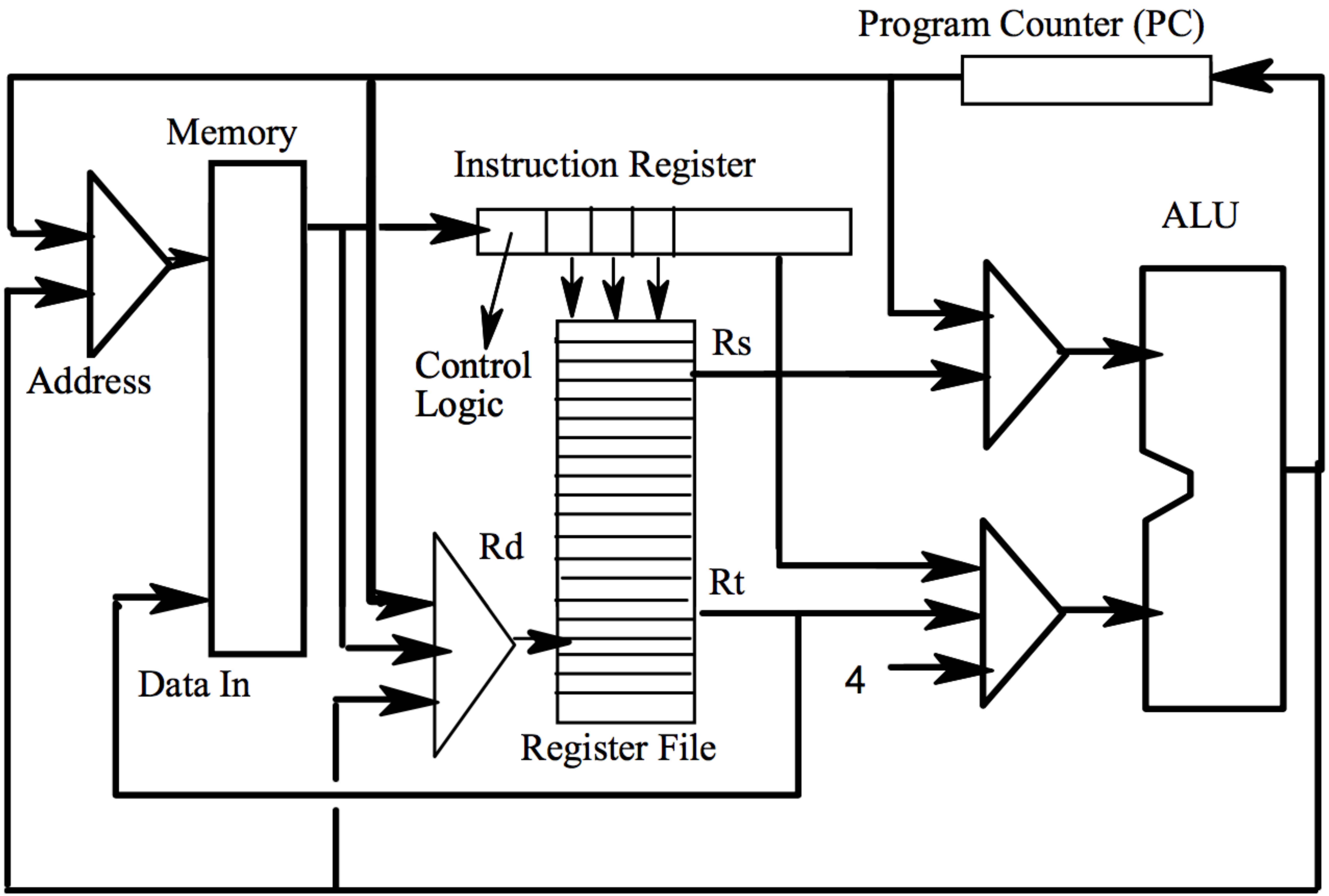
- So far, we've built a generic machine
- Now specify to MIPS
  - Microprocessor without Interlocking Pipeline Stages
- Popular embedded processor
  - Has been used in computers, but currently primarily in embedded devices
- Devices using MIPS:
  - TV Set-top-boxes; VOIP handsets; DVD recorders; cable modems and wireless routers; Digital Cameras; printers and copiers; external hard drives; Sony PSP

simple MIPS architecture



- All our same pieces
  - New pieces we'll discuss later are circled
    - *Control signals* are derived from instruction

Britton's version (less pretty)



Main difference in Britton's version

- Only one memory which contains both data and instructions
  - more like real life, in some ways
  - more details in CS 301
  - called ***Von Neumann*** architecture
- Our version has separate memory for instructions and data
  - More like on-chip cache
  - more details in CS 301
  - called ***Harvard*** architecture

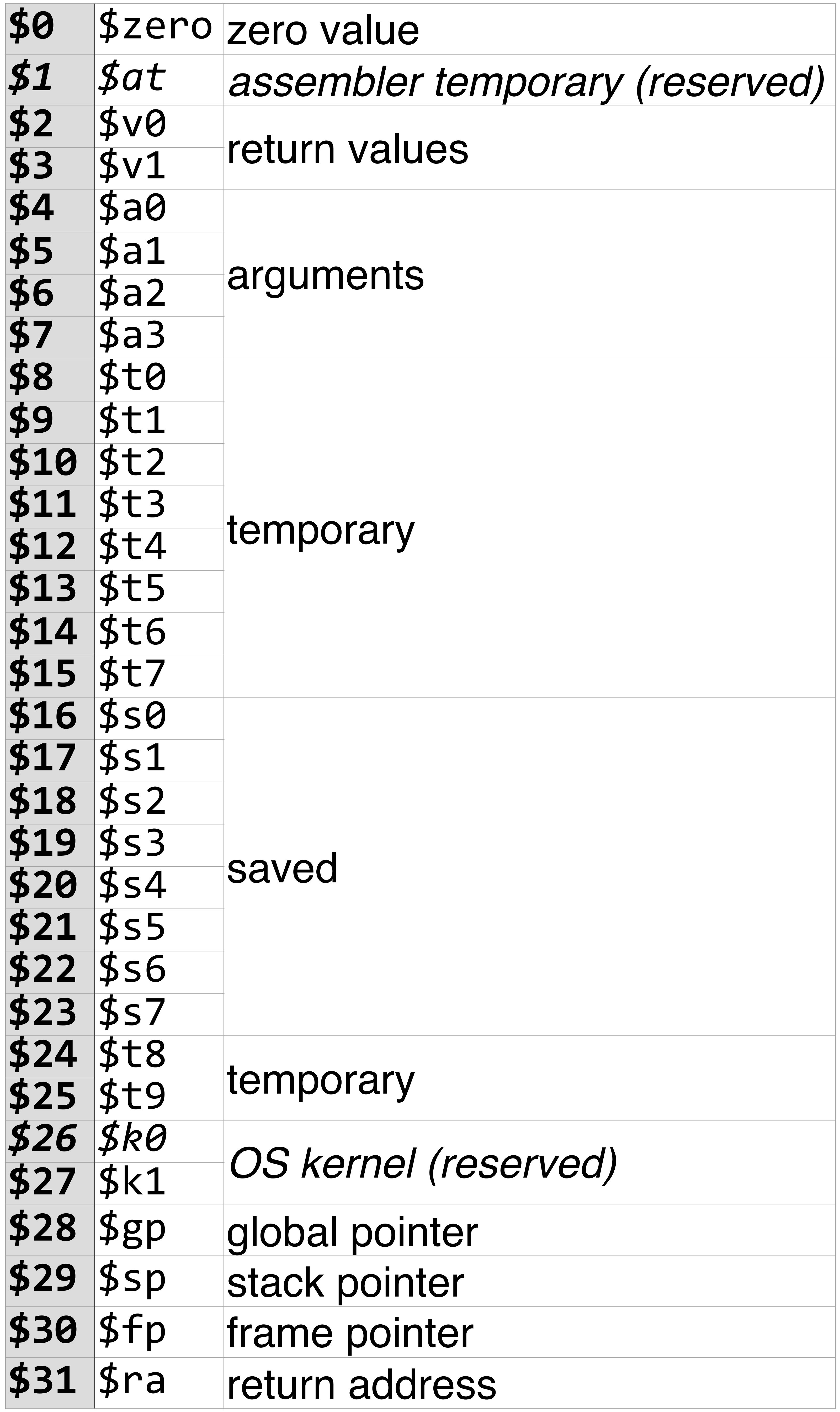
MIPS processor architecture

- Don’t get intimidated
- We’ve built all of these things already
- All with more details in CS 301
  - Memory
  - Multiplexers
  - Registers and Register file
  - ALU
  - Control Logic is just combinational logic
    - input = instruction opcode
    - output = control signals to multiplexers, ALU etc

MIPS register file

- 32 registers labeled $0 through $31
  - so we need 5 bits of the instruction to specify each register we will be using
- Each register also has special purpose functions and names

MIPS register file



$0	$zero	zero value
$1	*$at*	*assembler temporary (reserved)*
$2	$v0	return values
$3	$v1	
$4	$a0	arguments
$5	$a1	
$6	$a2	
$7	$a3	
$8	$t0	temporary
$9	$t1	
$10	$t2	
$11	$t3	
$12	$t4	
$13	$t5	
$14	$t6	
$15	$t7	
$16	$s0	saved
$17	$s1	
$18	$s2	
$19	$s3	
$20	$s4	
$21	$s5	
$22	$s6	
$23	$s7	
$24	$t8	temporary
$25	$t9	
$26	*$k0*	*OS kernel (reserved)*
$27	$k1	
$28	$gp	global pointer
$29	$sp	stack pointer
$30	$fp	frame pointer
$31	$ra	return address

$0	$zero	zero value
$1	*$at*	*assembler temporary (reserved)*
$2 - $3	$v0 - $v1	return values
$4 - $7	$a0 - $a4	arguments
$8 - $15	$t0 - $t7	temporary
$16 - $23	$s0 - $s7	saved
$24 - $25	$t8 - $t9	temporary
$26 - $27	*$k0 - $k1*	*OS kernel (reserved)*
$28	$gp	global pointer
$29	$sp	stack pointer
$30	$fp	frame pointer
$31	$ra	return address

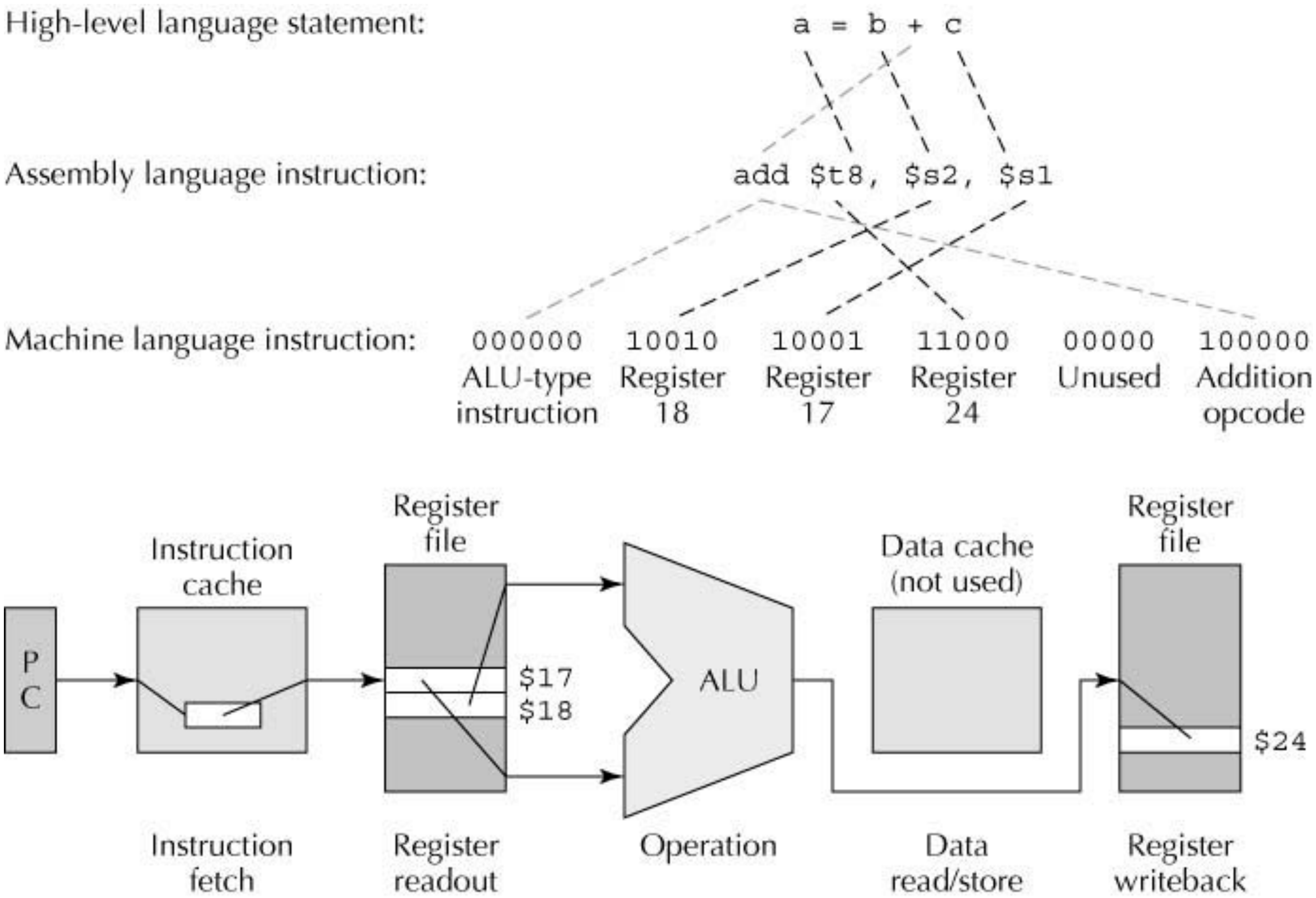
Memory structure

- Most memory is physically 8 bits wide
- MIPS words (single instruction or piece of data) are typically 32 bits wide
- Words take 4 addresses in memory
- Some data can take more or fewer slots
  - Byte = 1 address
  - Double word = 8 addresses
- Note: “word” is usually specific to the computer system
  - 16 bits, 32 bits, newer 64-bit systems

Memory structure

- MIPS is “big-endian”
  - Most significant byte is in lower memory address
- x86 is “little-endian”
  - Least significant byte is in lower memory address
  - One cause of incompatibility between machines
- Note: Book uses “cache” in place of memory sometimes
  - For our purposes, Cache = memory
  - Cache is a small fast area of memory holding recently used or soon to be used data or instructions.

Example MIPS instruction



High-level language statement: a = b + c

Assembly language instruction: add $t8, $s2, $s1

Machine language instruction:

000000	10010	10001	11000	00000	100000
ALU-type instruction	Register 18	Register 17	Register 24	Unused	Addition opcode

Example MIPS instruction

- Note: two register files??
  - Not really two register files
  - This “unwrapped” representation is intended to make the diagram clearer
  - Separates “register read” from “register writeback”
  - more clearly indicates how long these things take
    - Load an instruction from memory, which selects multiplexer lines, which enables reading from registers, which are propagated to ALU inputs, which are fed through an adder, whose result is sent to a register input
    - all before the next clock cycle…

Machine Language

- The MIPS computer doesn’t see

```
add $t8, $s2, $s1
```

- All it sees is

```
000000 10010 10001 11000 00000 100000
```

  - Stored in the instruction register
  - These bits tell the computer what to do and when to make the addition happen
  - Not useful for us humans

Aside: Recall HEX encodings (we'll be using them)

- 0b0000 = 0
- 0b1000 = 8
- 0b1010 = A
- 0b1111 = F
- 0b110000 = ?
- 0b1010101 = ?

C = ?

B = ?

E = ?

4 = ?

0xFACE = ?

0xDB9 = ?

0xDEADBEEF = ?

0x = hex indicator

0b = binary indicator

0o = octal indicator

Assembly Language

- When you code in a High Level Language
  - Compiler translates this into assembly language
  - which is then translated into Machine code
- Machine Code
  - strings of 0s and 1s which tell the machine what to do
  - Good for computers, hard for humans
    - acres and acres of 1s and 0s

Assembly Language

- One step up from machine language
  - Human-readable
  - direct 1-to-1 correspondence with machine language
  - (almost) every assembly language instruction can be translated into a single machine language instruction
    - some take more than one machine instruction
  - We write in assembly language, and an assembler translates it into machine language

Why Learn Assembly Language

- Historical context
- So we can write better HLL code
  - we have an idea what is actually going on when we execute HLL things
- So we can actually code in AL
  - *Very* fast and space-efficient code
  - *Direct* control over almost all computer behaviour
  - Access to system calls and behaviour

So why not code in assembly language all the time

- because it's *really* hard to make big complex programs
  - think data mining
- because it's not programmer-efficient
  - think HTML vs Wordpress
- because it's not platform independent
  - although native code is always better than cross-compiled crap.

Two types of assembly statements

- Instructions
  - translated into machine code
  - Also macro-instructions
    - Translated into a few normal instructions
- Assembler Directives
  - tells the assembler what do to, sets parameters, allocates data etc.
  - Proceeded with a “.”

Structure of assembly statements

```
{label:}  Keyword  {parameter(s)} {# comment}
```

- ***label***: identifier for assembler to make connections between statements
  - Good practice to have labels on a separate line
- ***Keyword***: the identifier for the statement
  - Instruction opcode or directive name
- ***Parameter***: control the behaviour of the statement
- ***Comment***: for us poor humans to understand

Examples

- Instruction

```
ADD $a1, $a2, $a3
```

tells the computer to take the values in `$a2` and `$a3`, add them, and put the result in `$a1`

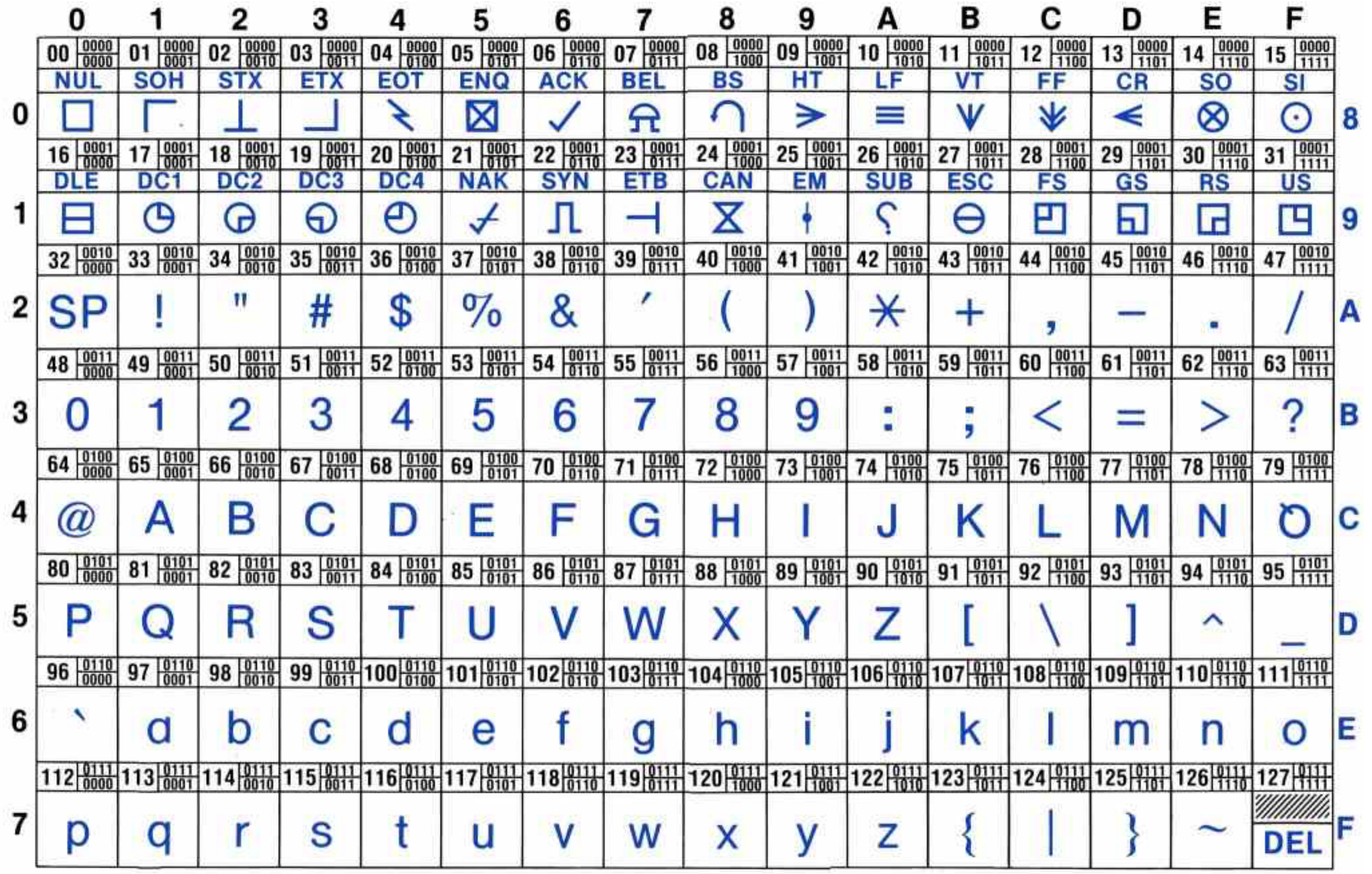
- Directive

```
.asciiz “foo”
```

while assembling, stores the ascii values of the letters `f`, `o`, `o` in the next three locations in memory

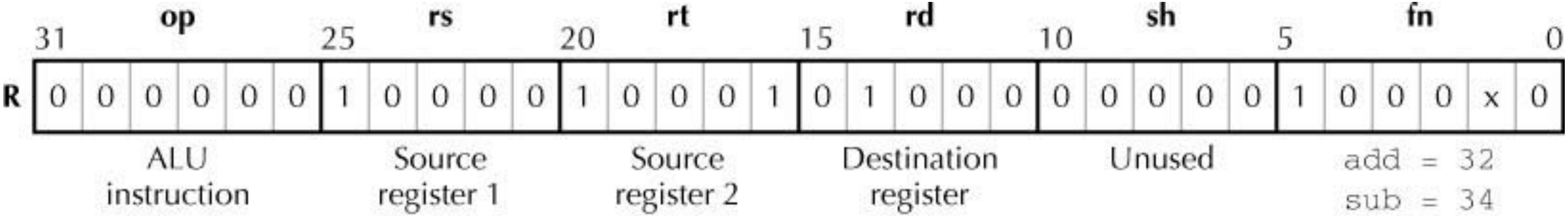
aside: ASCII

- American Standard Code for Information Interchange
  - 7-bit (1 word) code for characters
    - letters, numbers, symbols etc
    - 8 bits for extended ascii
    - more bits for unicode
  - 100 0001 = 65_{10} = 0x41 = ‘A’
  - 110 0001 = 97_{10} = 0x61 = ‘a’
  - 110 0010 = 98_{10} = 0x62 = ‘b’
  - 011 0000 = 48_{10} = 0x30 = ‘0’
  - 011 1001 = 57_{10} = 0x39 = ‘9’



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	00 0000 0000 NUL	01 0000 0001 SOH	02 0000 0010 STX	03 0000 0011 ETX	04 0000 0100 EOT	05 0000 0101 ENQ	06 0000 0110 ACK	07 0000 0111 BEL	08 0000 1000 BS	09 0000 1001 HT	10 0000 1010 LF	11 0000 1011 VT	12 0000 1100 FF	13 0000 1101 CR	14 0000 1110 SO	15 0000 1111 SI	8
1	16 0001 0000 DLE	17 0001 0001 DC1	18 0001 0010 DC2	19 0001 0011 DC3	20 0001 0100 DC4	21 0001 0101 NAK	22 0001 0110 SYN	23 0001 0111 ETB	24 0001 1000 CAN	25 0001 1001 EM	26 0001 1010 SUB	27 0001 1011 ESC	28 0001 1100 FS	29 0001 1101 GS	30 0001 1110 RS	31 0001 1111 US	9
2	32 0010 0000 SP	33 0010 0001 !	34 0010 0010 "	35 0010 0011 #	36 0010 0100 $	37 0010 0101 %	38 0010 0110 &	39 0010 0111 '	40 0010 1000 (	41 0010 1001)	42 0010 1010 *	43 0010 1011 +	44 0010 1100 ,	45 0010 1101 –	46 0010 1110 .	47 0010 1111 /	A
3	48 0011 0000 0	49 0011 0001 1	50 0011 0010 2	51 0011 0011 3	52 0011 0100 4	53 0011 0101 5	54 0011 0110 6	55 0011 0111 7	56 0011 1000 8	57 0011 1001 9	58 0011 1010 :	59 0011 1011 ;	60 0011 1100 <	61 0011 1101 =	62 0011 1110 >	63 0011 1111 ?	B
4	64 0100 0000 @	65 0100 0001 A	66 0100 0010 B	67 0100 0011 C	68 0100 0100 D	69 0100 0101 E	70 0100 0110 F	71 0100 0111 G	72 0100 1000 H	73 0100 1001 I	74 0100 1010 J	75 0100 1011 K	76 0100 1100 L	77 0100 1101 M	78 0100 1110 N	79 0100 1111 O	C
5	80 0101 0000 P	81 0101 0001 Q	82 0101 0010 R	83 0101 0011 S	84 0101 0100 T	85 0101 0101 U	86 0101 0110 V	87 0101 0111 W	88 0101 1000 X	89 0101 1001 Y	90 0101 1010 Z	91 0101 1011 [	92 0101 1100 \	93 0101 1101]	94 0101 1110 ^	95 0101 1111 _	D
6	96 0110 0000 `	97 0110 0001 a	98 0110 0010 b	99 0110 0011 c	100 0110 0100 d	101 0110 0101 e	102 0110 0110 f	103 0110 0111 g	104 0110 1000 h	105 0110 1001 i	106 0110 1010 j	107 0110 1011 k	108 0110 1100 l	109 0110 1101 m	110 0110 1110 n	111 0110 1111 o	E
7	112 0111 0000 p	113 0111 0001 q	114 0111 0010 r	115 0111 0011 s	116 0111 0100 t	117 0111 0101 u	118 0111 0110 v	119 0111 0111 w	120 0111 1000 x	121 0111 1001 y	122 0111 1010 z	123 0111 1011 {	124 0111 1100 |	125 0111 1101 }	126 0111 1110 ~	127 0111 1111 DEL	F

MIPS instruction format: Add and Subtract



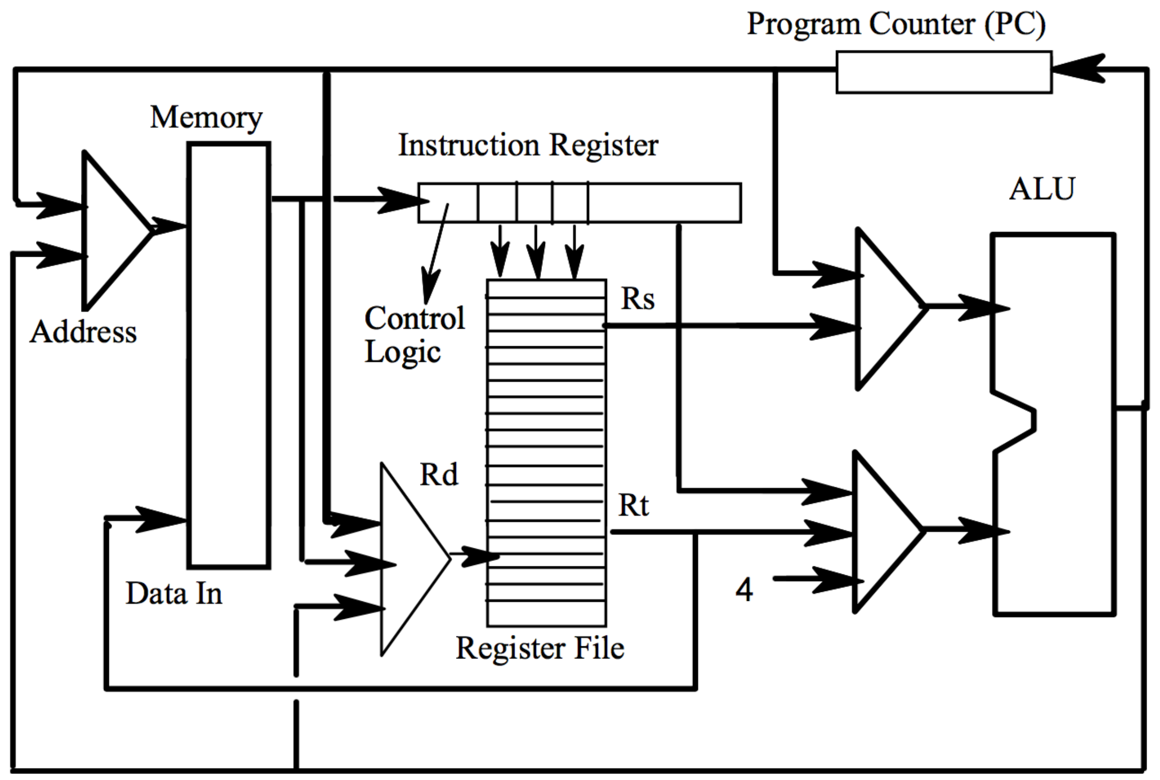
	op	rs	rt	rd	sh	fn
	31	25	20	15	10	5 0
R	0 0 0 0 0 0	1 0 0 0 0	1 0 0 0 1	0 1 0 0 0	0 0 0 0 0	1 0 0 0 x 0
	ALU instruction	Source register 1	Source register 2	Destination register	Unused	add = 32 sub = 34

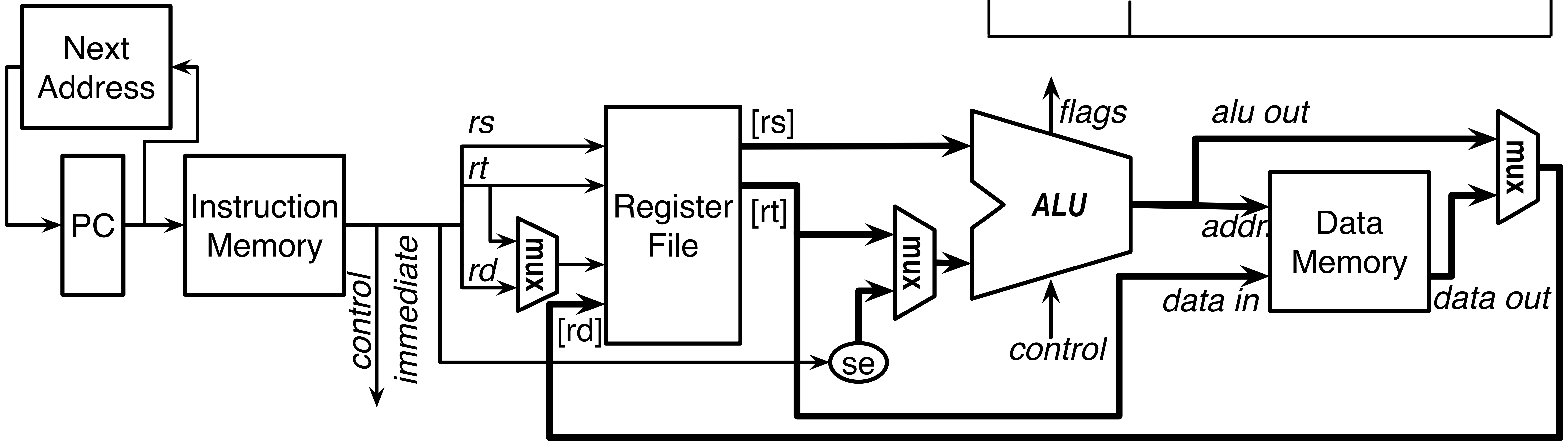
- op = opcode, different for each operation class
  - 000000 indicates an ALU operation
- rs, rt, rd indicate which registers to use
- sh is unused in add and subtract instructions
- fn indicates which specific operation to do
  - 100000 = add; 100010 = subtract

Register format instructions

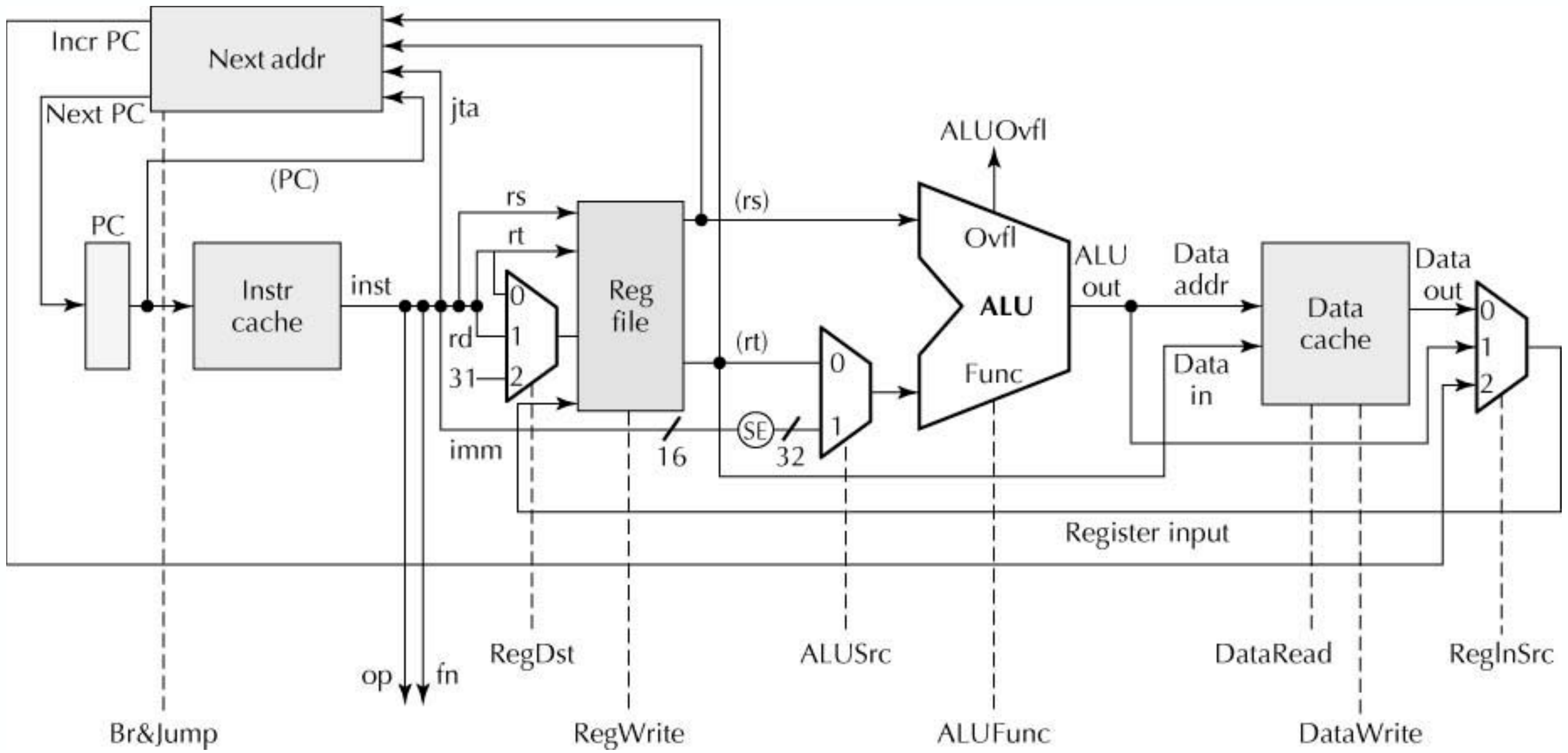
- e.g. add and subtract
- All information is coded in registers
  - rs and rt are usually source registers
  - rd is usually the destination register
  - sh is the shift amount (we'll see soon)
- Function code (fn) extends opcode (op)
  - More possible operations than the 64 available in a 6-bit number

Aside: CPU figures



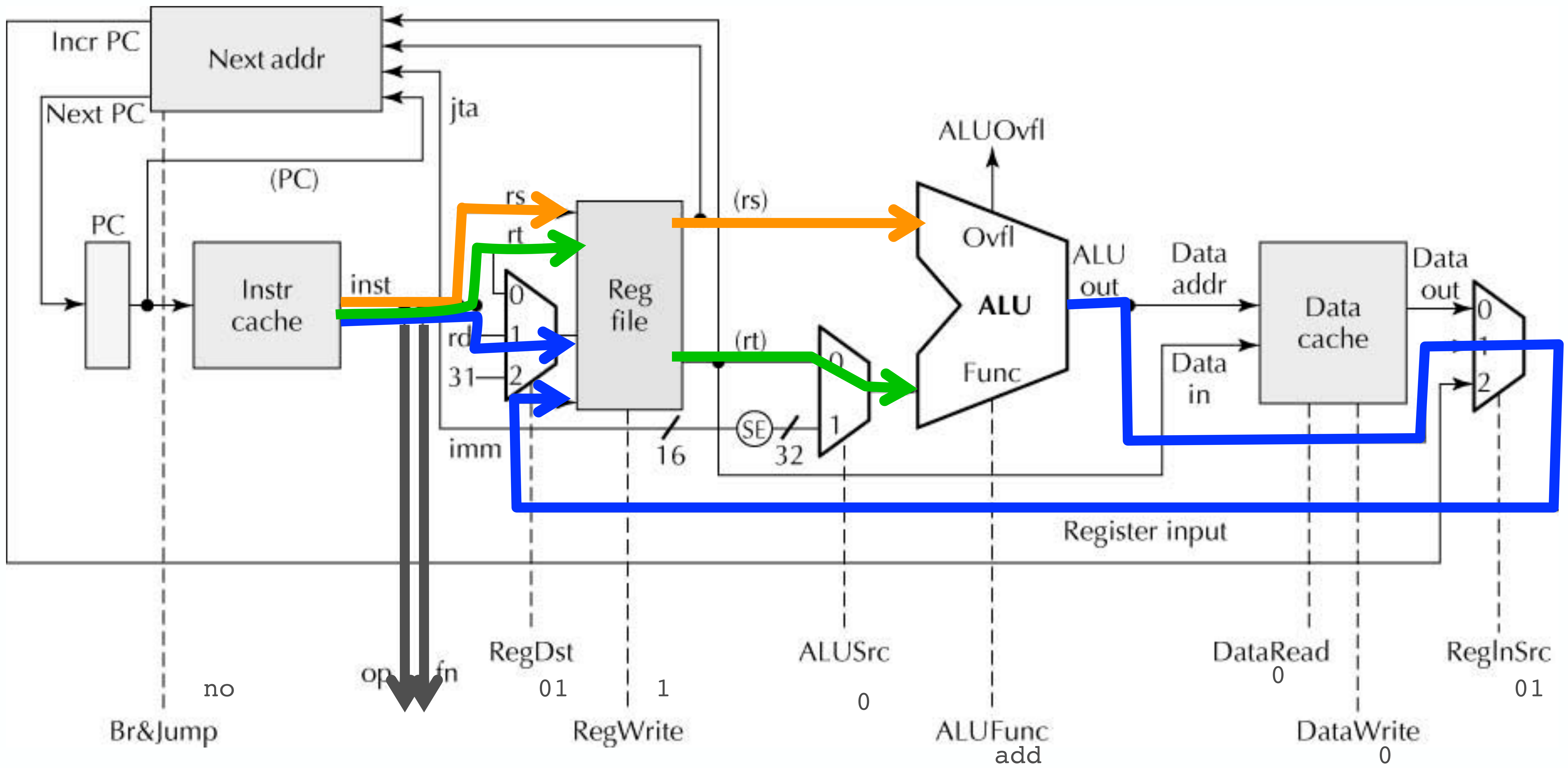


- Some of these figures are from a different text, some I've put together myself



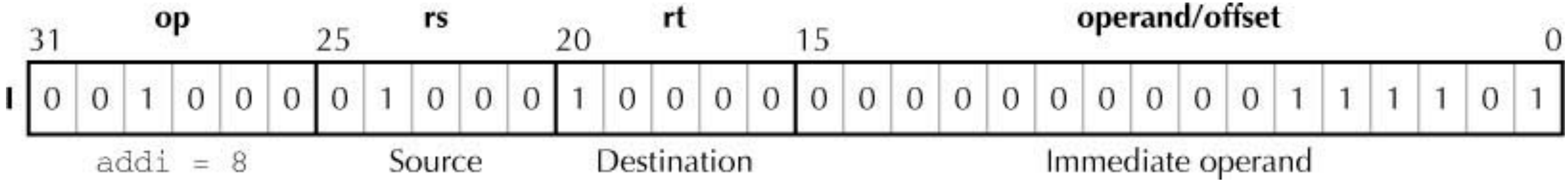
- Same concept

Register: $add $t8, $s2, $s1



Immediate instructions

- To add a constant, hard-code the value into the instruction itself
  - Easier than computing it, storing it in memory, then loading it into a register
- Called “immediate”
  - value is immediately available in the instruction

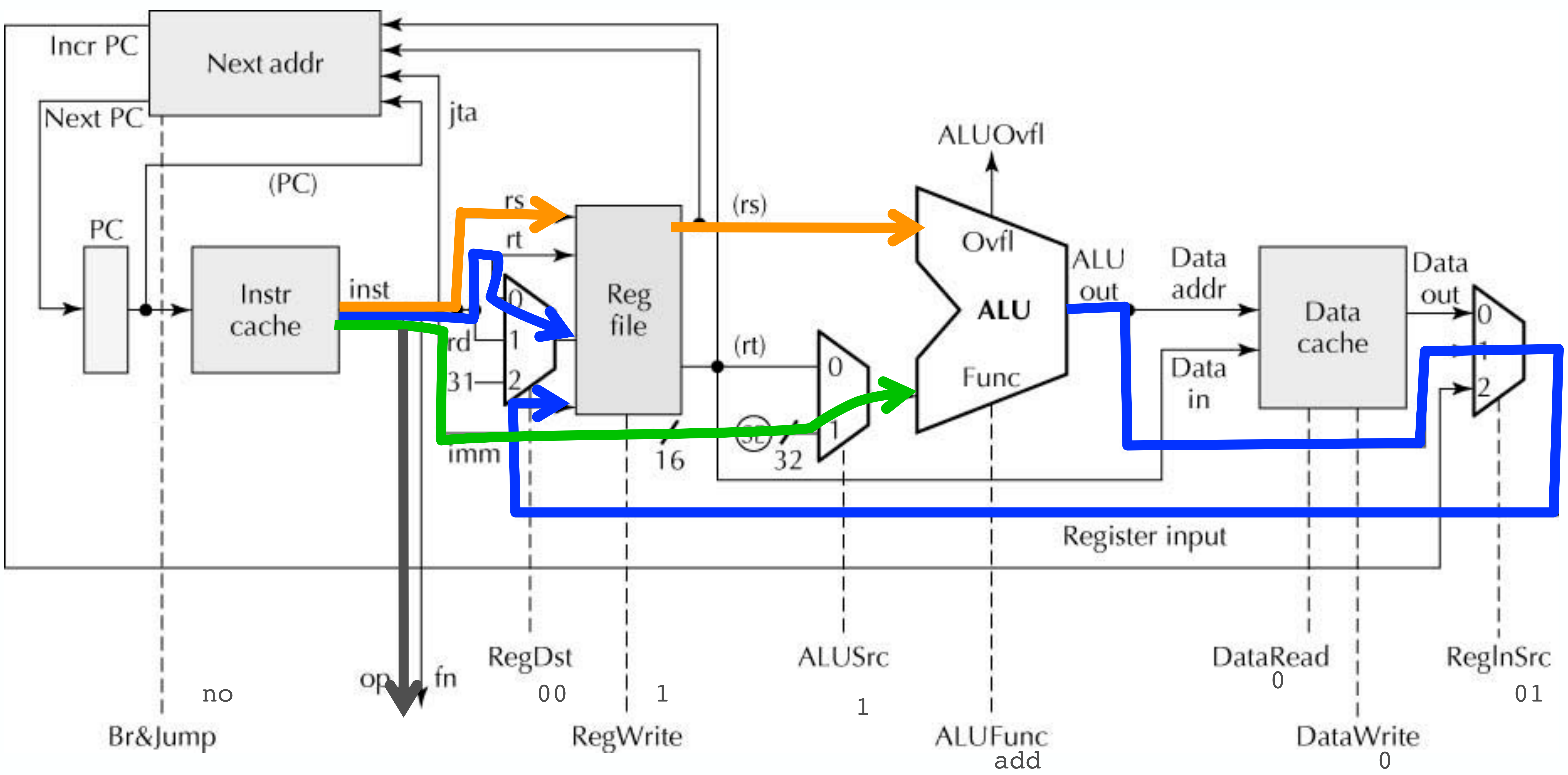


	op	rs	rt	operand/offset
	31	25	20	15 0
I	0 0 1 0 0 0	0 1 0 0 0	1 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1
	addi = 8	Source	Destination	Immediate operand

Immediate instructions

- No function code
  - Opcode completely specifies the action
- ***rt*** is now the ***destination***, and ***rs*** is a source
  - In register-format instructions, rs and rt are source registers and rd is the destination register
  - in immediate-format instructions, rs is the source, rt is the destination

Immediate: `addi $t1,$s0,61`



Immediate: se?

- Sign Extension
  - Immediate field is only 16 bits long
  - Adder is 32 bits
    - needs a 32-bit number
  - Sign-extend fills the top 16 bits with zeros (if positive) or 1s (if negative)
    - doesn’t change the value, maintains the sign
    - Sign is replicated to fill upper bits

Load and Store

- Transfer data from the memory to a register, or from a register to memory
  - Requires the register and the memory address
  - Transfers 4 bytes at a time (Registers are 32 bits)
- Location in memory is specified by two pieces of information: base plus offset
  - ***Base*** is in the register specified by *base register*
  - ***Offset*** is specified as an *immediate* value
  - actual address is calculated at run time by the ALU

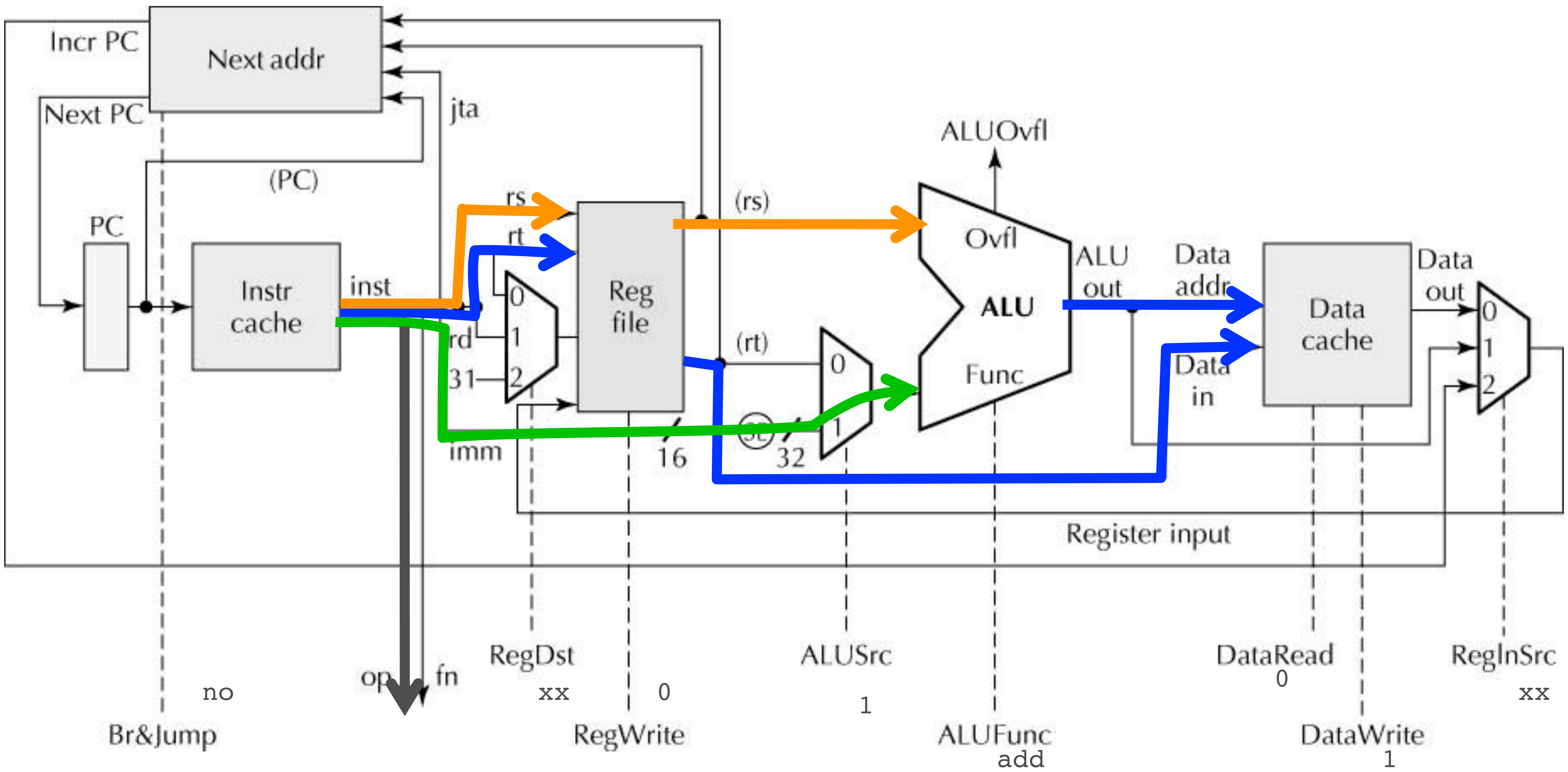
    - `lw $t0,40($s3)` # $t0 ← mem(40+$s3)

Load and Store

op
rs
rt
operand/offset
31
25
20
15
0
I
1 0 x 0 1 1
1 0 0 1 1
0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0
lw = 35
sw = 43
Base register
Data register
Offset relative to base

- Considered an immediate-format instruction
  - Offset is an immediate value
- No function code - opcode completely specifies load and store instructions.
  - Either base or offset can be set to "0" for more flexibility in addressing data.

Load and Store: `sw $t0,40($s3)`



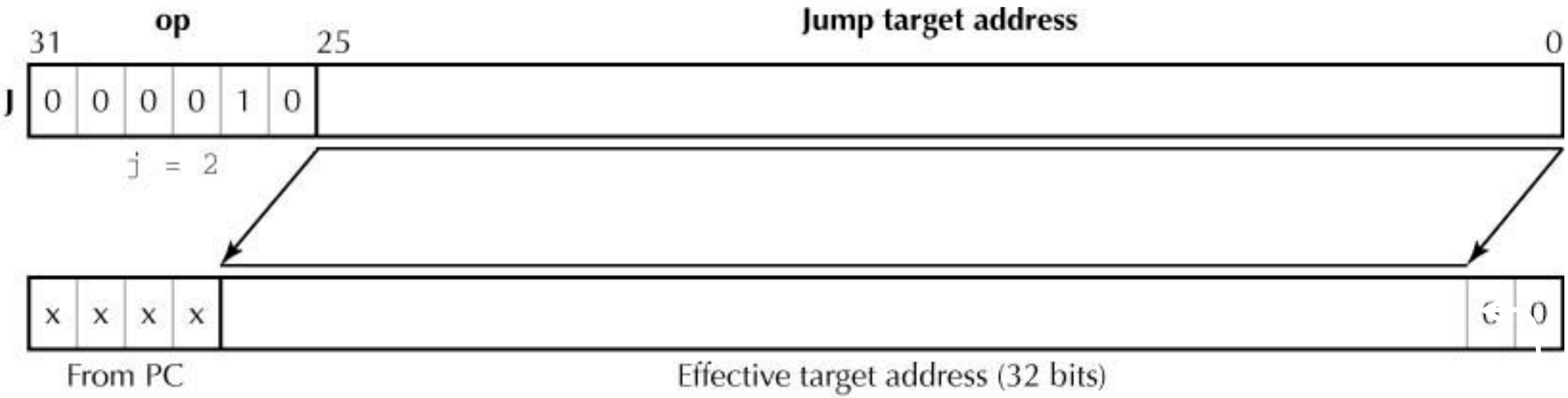
Program flow

- Normally, instructions are stored sequentially
  - 32 bits long, so 4 memory addresses per.
  - do this, then do that.
- Occasionally, we will want to execute a different instruction
  - from a different memory address
  - branches and jumps
  - we'll do jumps now, branches later

Jump instructions (`j`)

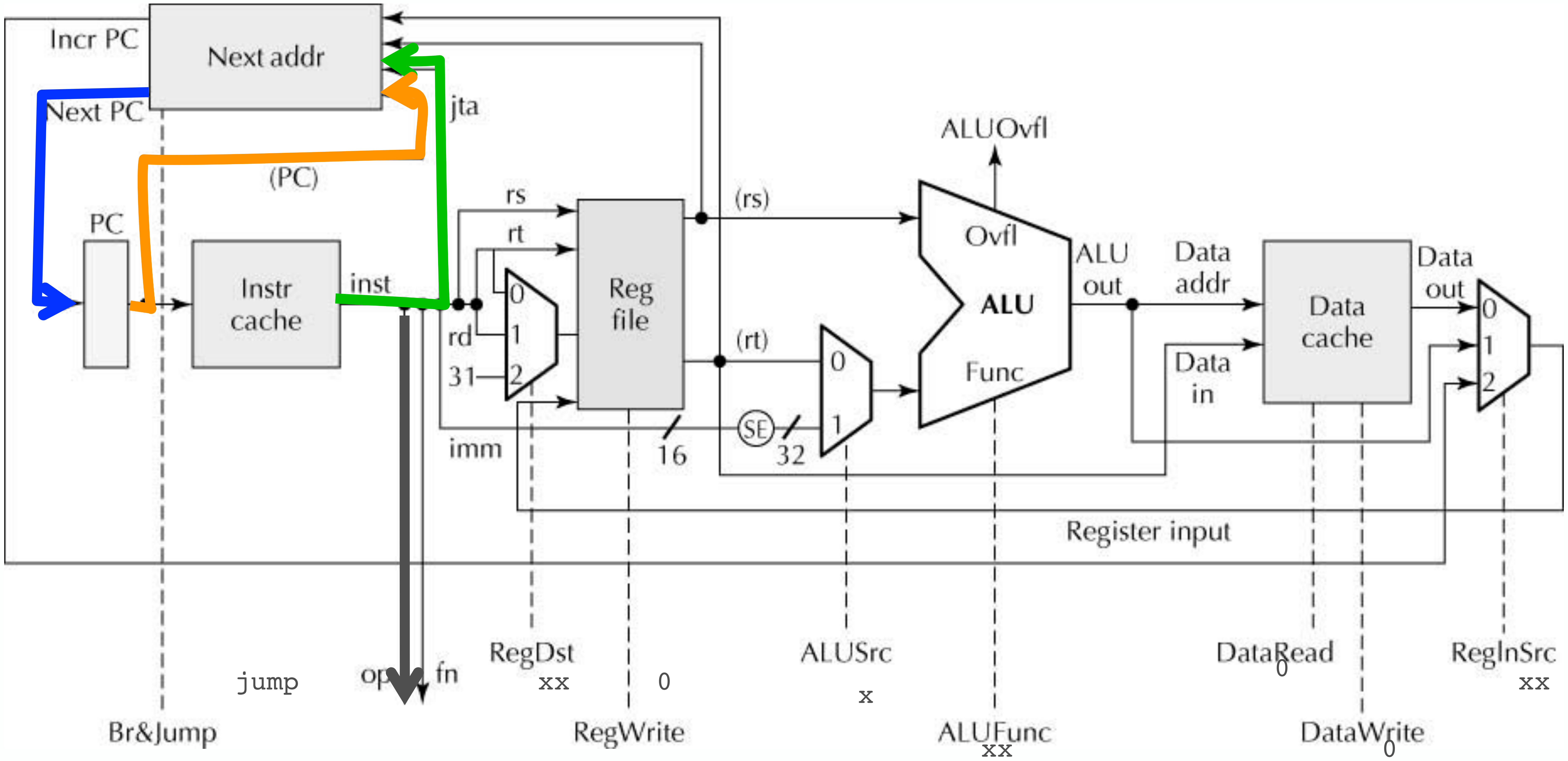
- Used to change the value of the program counter
- Recall: Program counter indicates where in memory the next instruction will come from
- Normally, next instruction is 4 bytes after current instruction
- Jump instructions act like a “goto”
  - Next instruction will be somewhere else

Jump instructions

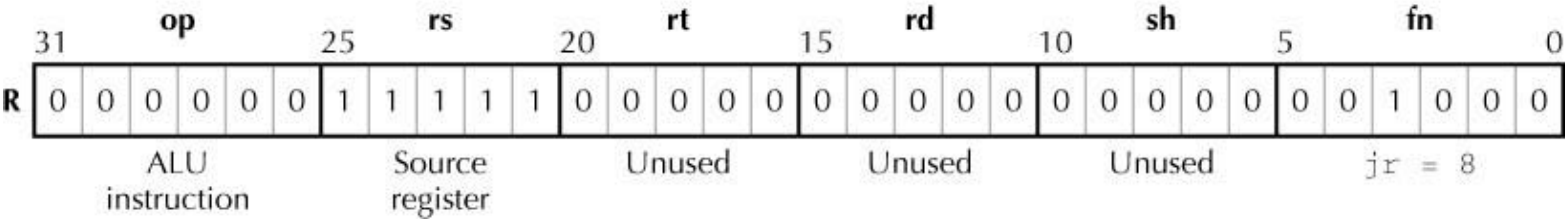


- New address loaded into PC
- instructions are 4 bytes long, so all instruction addresses end in 00
- Can’t quite go anywhere: top 4 bits are from previous PC value

Jump instructions



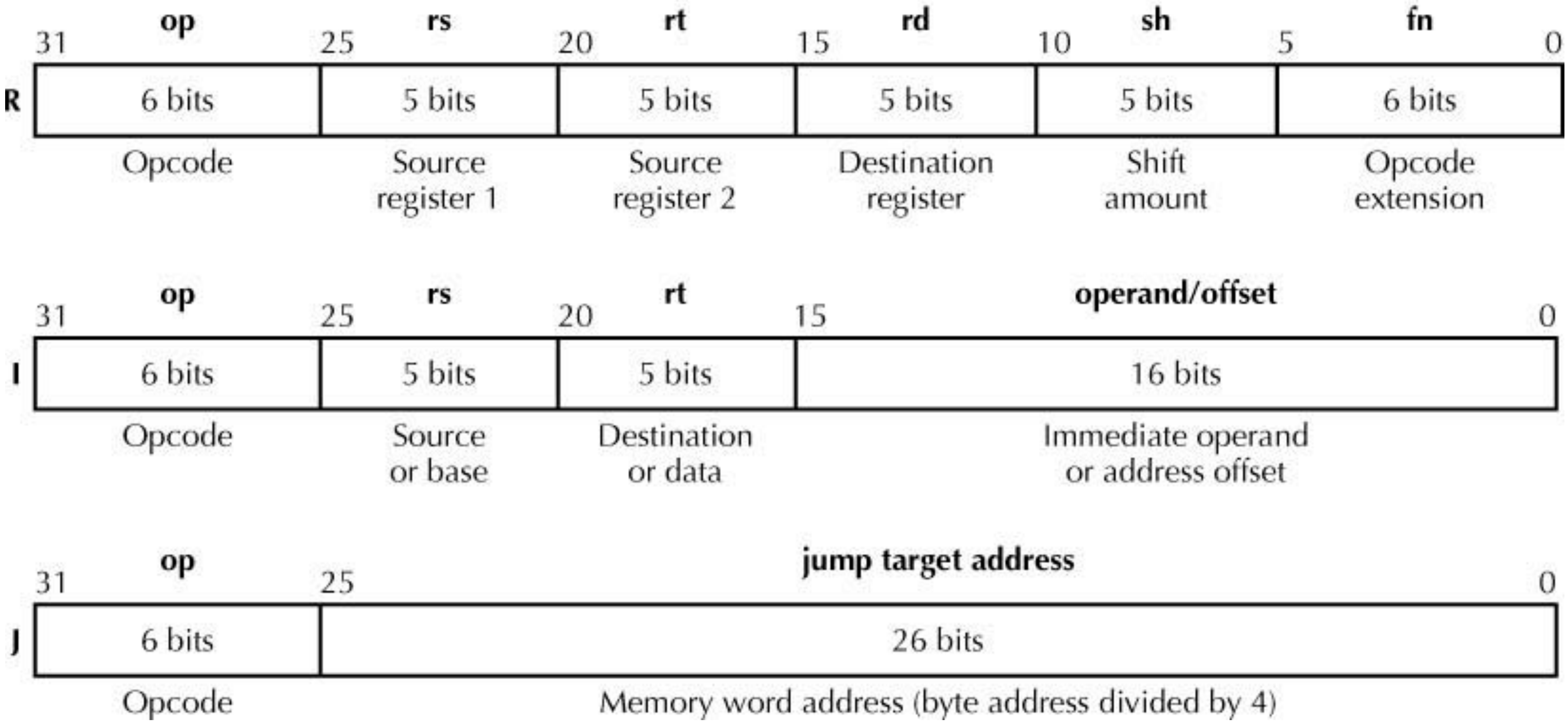
Register jump instructions (`jr`)



	op	rs	rt	rd	sh	fn
	31	25	20	15	10	5 0
R	0 0 0 0 0 0	1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 1 0 0 0
	ALU instruction	Source register	Unused	Unused	Unused	`jr` = 8

- Like a jump, but the destination is held in a register instead of an immediate value

Summary: instruction formats



- And that’s it! (for formats. Lots more language)

Summary: instruction formats

- All instructions in MIPS will be one of these formats
  - branch instructions use immediate format
  - Other instructions as we go
- Next section: more instructions, some simple assembly language programs, more hardware
  - What can this computer do?
- Specifics of assembly language addressing