Rechnerarchitektur

SCHNIITSTELLE

AUDIO BUCH

Agenda

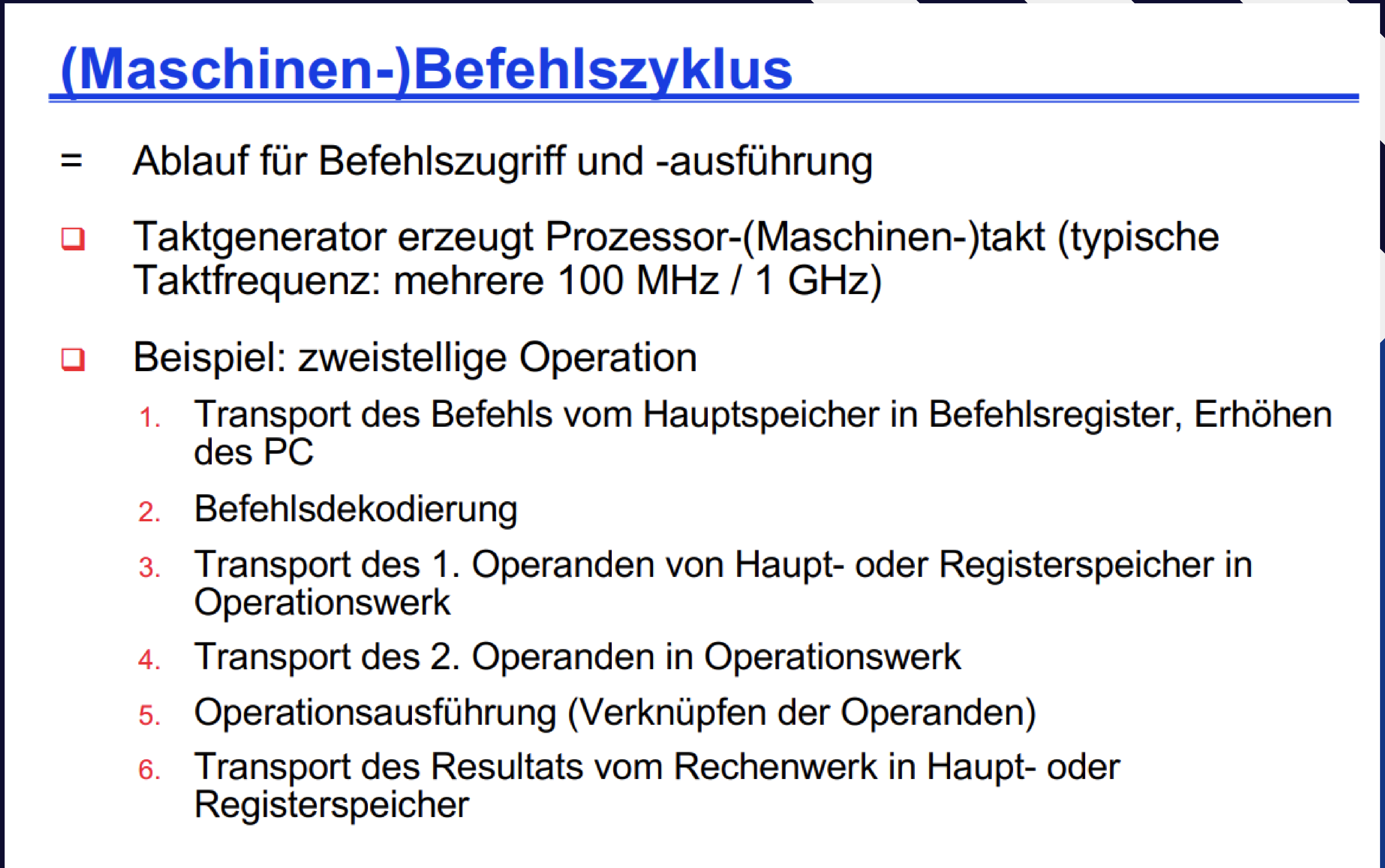
Topics Covered

Topic 1

Topic 2

Topic 3

Topic 4

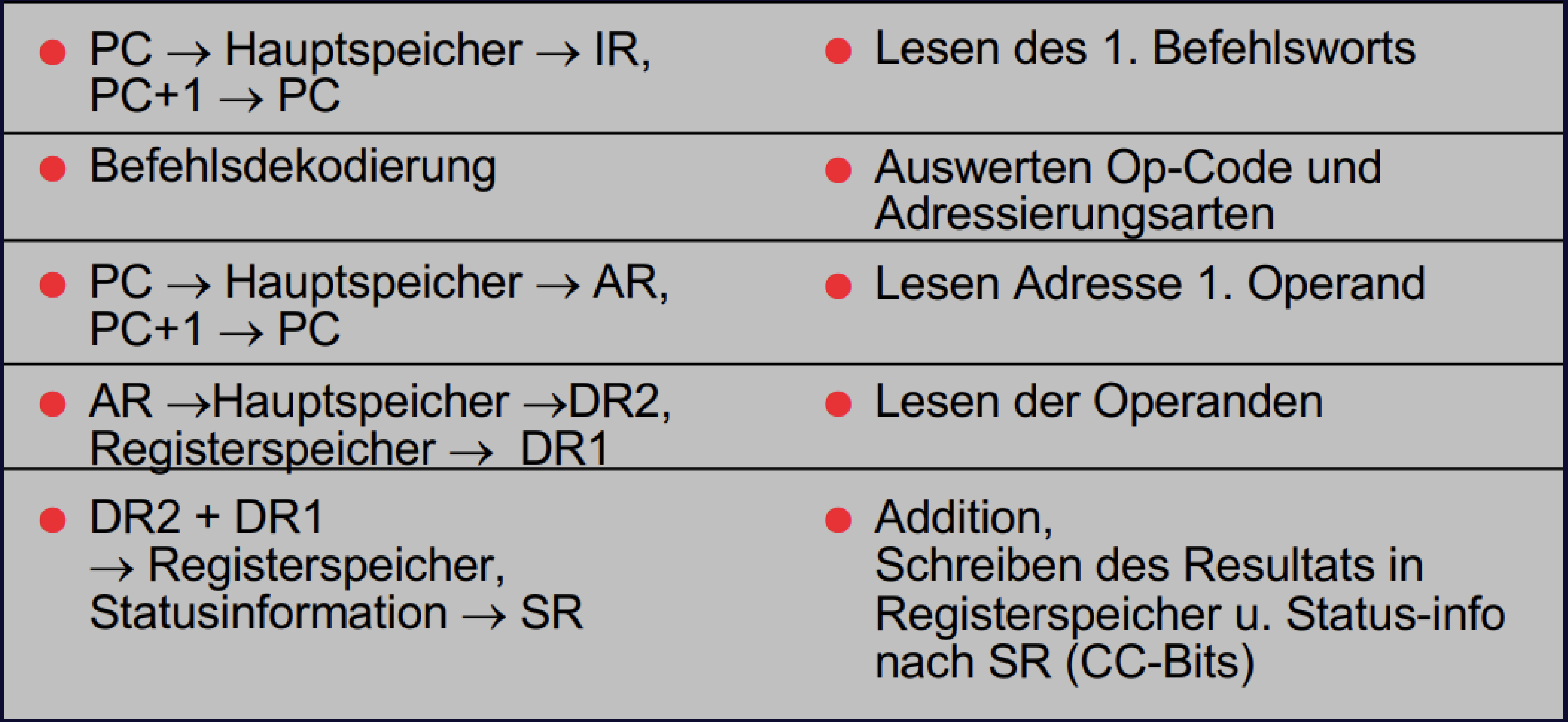


(Maschinen-)Befehlszyklus

= Ablauf für Befehlszugriff und -ausführung

- Taktgenerator erzeugt Prozessor-(Maschinen-)takt (typische Taktfrequenz: mehrere 100 MHz / 1 GHz)
- Beispiel: zweistellige Operation
  1. Transport des Befehls vom Hauptspeicher in Befehlsregister, Erhöhen des PC
  2. Befehlsdekodierung
  3. Transport des 1. Operanden von Haupt- oder Registerspeicher in Operationswerk
  4. Transport des 2. Operanden in Operationswerk
  5. Operationsausführung (Verknüpfen der Operanden)
  6. Transport des Resultats vom Rechenwerk in Haupt- oder Registerspeicher

Beispiel Befehlszyklus



● PC → Hauptspeicher → IR, PC+1 → PC	● Lesen des 1. Befehlsworts
● Befehlsdekodierung	● Auswerten Op-Code und Adressierungsarten
● PC → Hauptspeicher → AR, PC+1 → PC	● Lesen Adresse 1. Operand
● AR →Hauptspeicher →DR2, Registerspeicher → DR1	● Lesen der Operanden
● DR2 + DR1 → Registerspeicher, Statusinformation → SR	● Addition, Schreiben des Resultats in Registerspeicher u. Status-info nach SR (CC-Bits)

ADD SPADR, R5

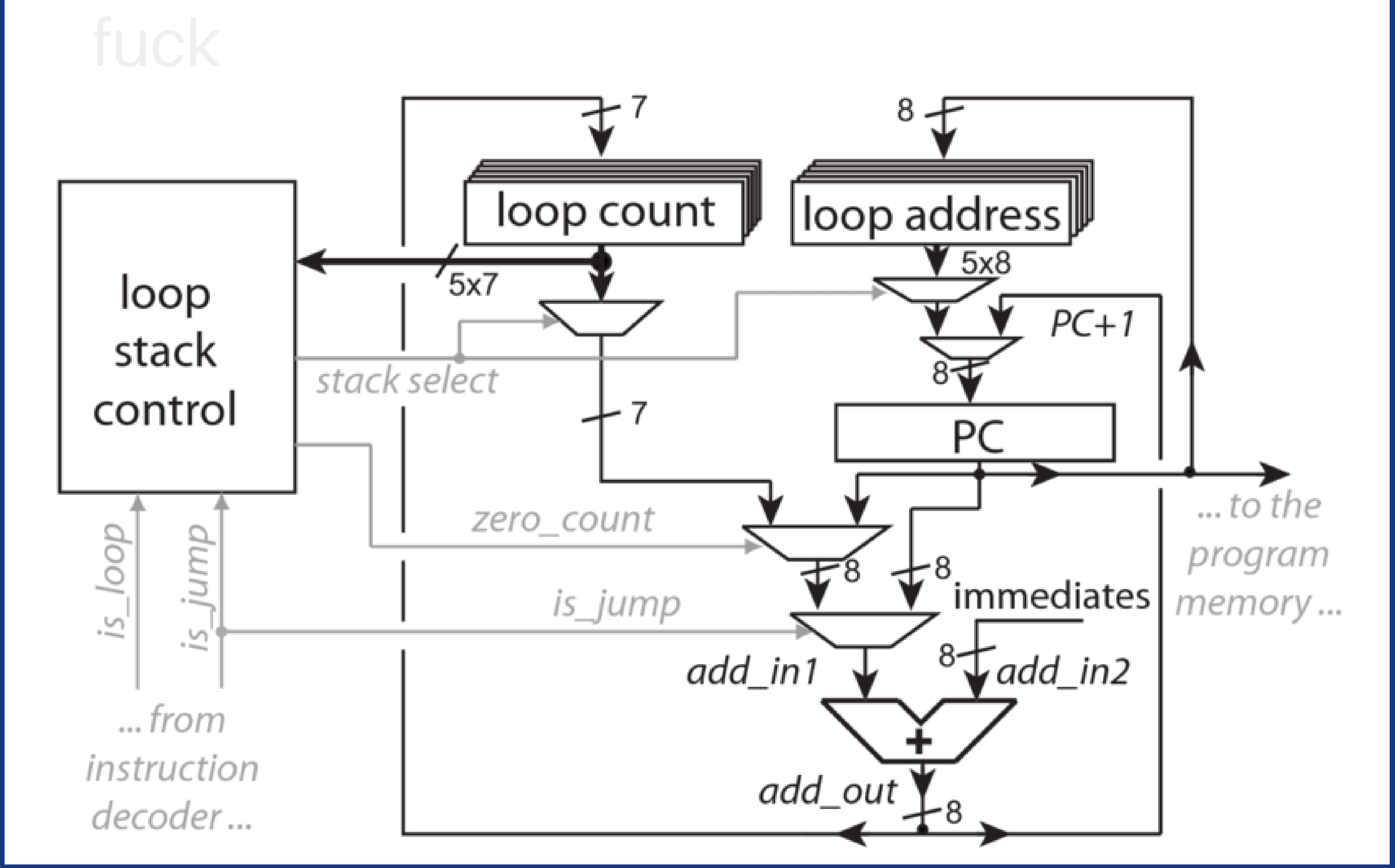
CISC-Mikroprozessor

- in den 70er Jahren: Ausstattung der Prozessoren mit immer mächtigeren Befehlssätzen
  - Ziel: Verringern der semantischen Lücke zwischen höheren Programmiersprachen und einfachen Maschinenbefehlen
- typisch: > 200 Befehle
- grosse Anzahl von Adressierungsarten
- viele Kombinationen von Befehlen und Adressierungsarten
- Mikrocode für jeden Befehl in Steuerwerk
- Mikroprogrammierung des Steuerwerks ist langsamer als feste Verdrahtung.
- Versuch, CPU durch komplexe Instruktionen stärker zu belasten (Speicherbus als Flaschenhals)

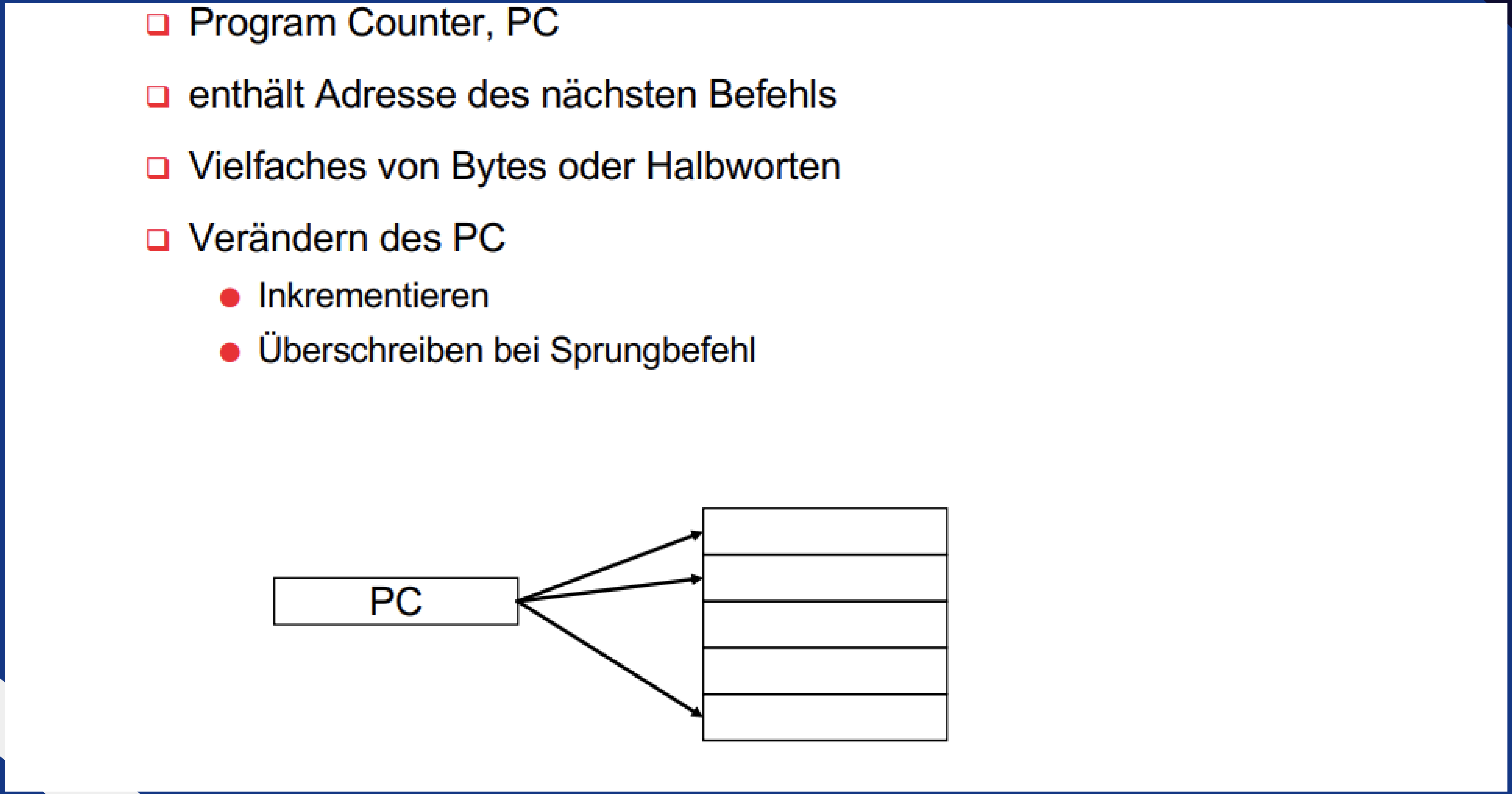
RISC-Mikroprozessor

fuck

RISC-Mikroprozessor



Program Counter, PC

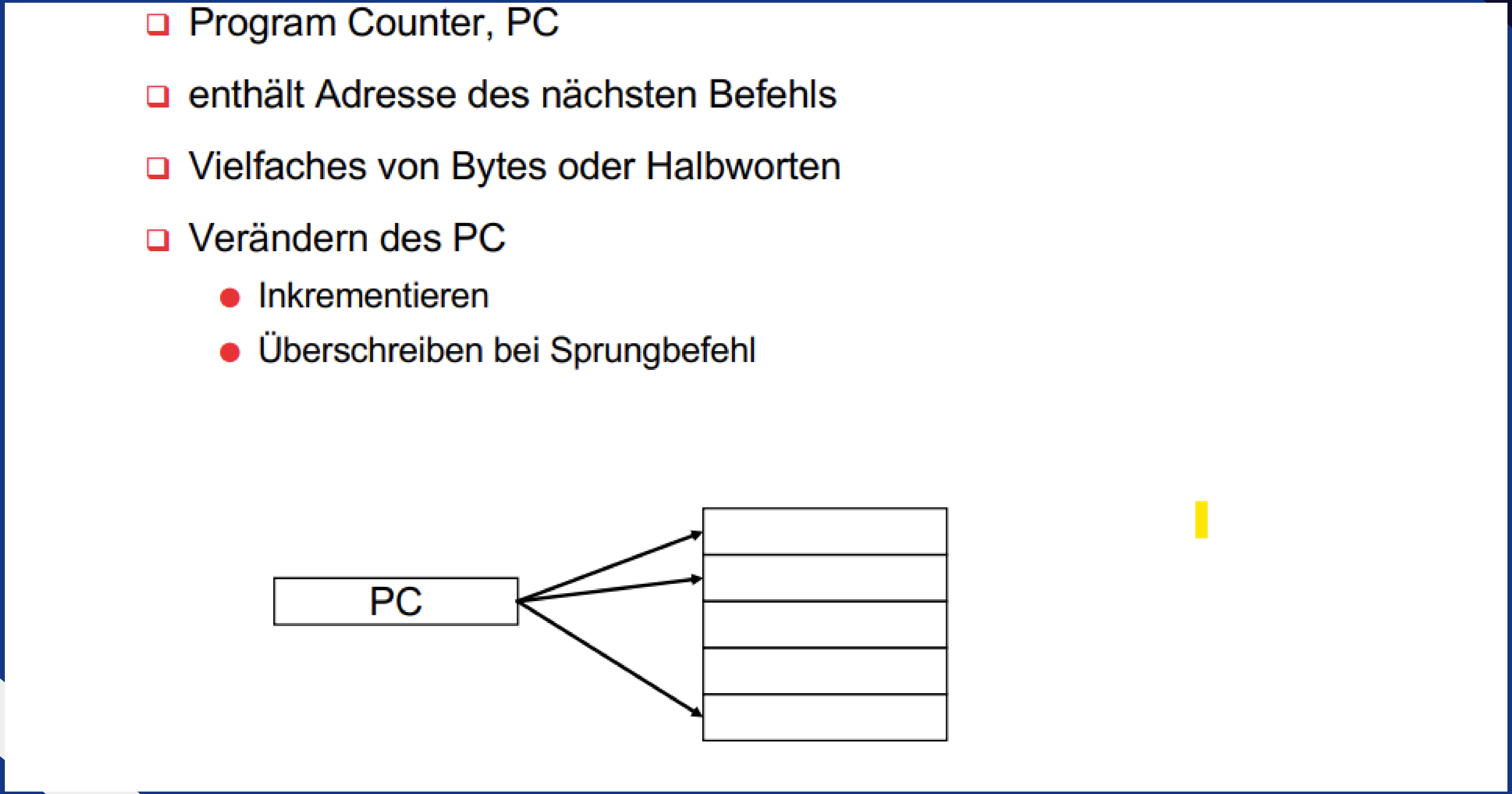


- Program Counter, PC
- enthält Adresse des nächsten Befehls
- Vielfaches von Bytes oder Halbworten
- Verändern des PC
  - Inkrementieren
  - Überschreiben bei Sprungbefehl

PC?

Go Back to Agenda Page

Program Counter, PC

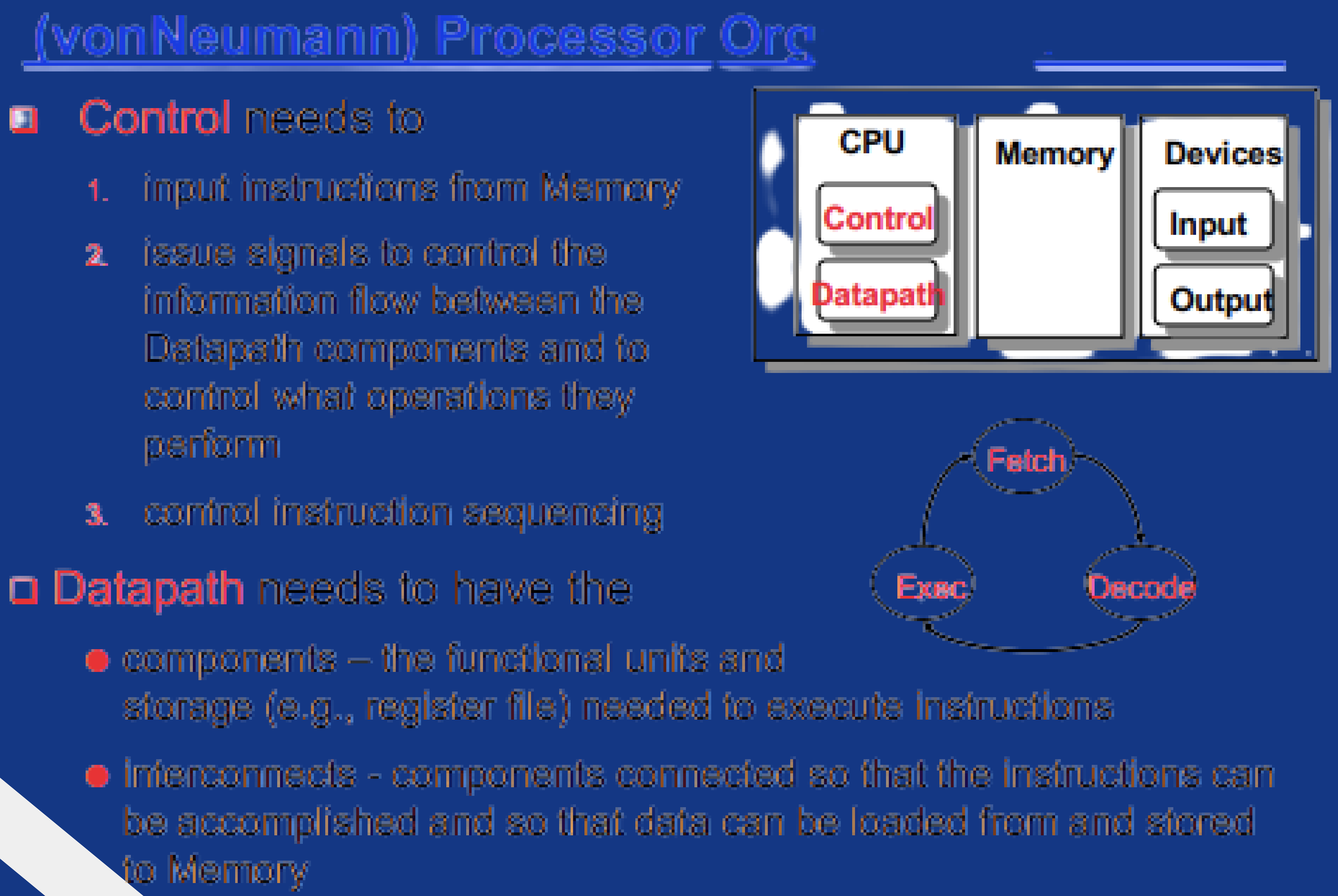


- ❑ Program Counter, PC
- ❑ enthält Adresse des nächsten Befehls
- ❑ Vielfaches von Bytes oder Halbworten
- ❑ Verändern des PC
  - ● Inkrementieren
  - ● Überschreiben bei Sprungbefehl

PC?

Go Back to Agenda Page

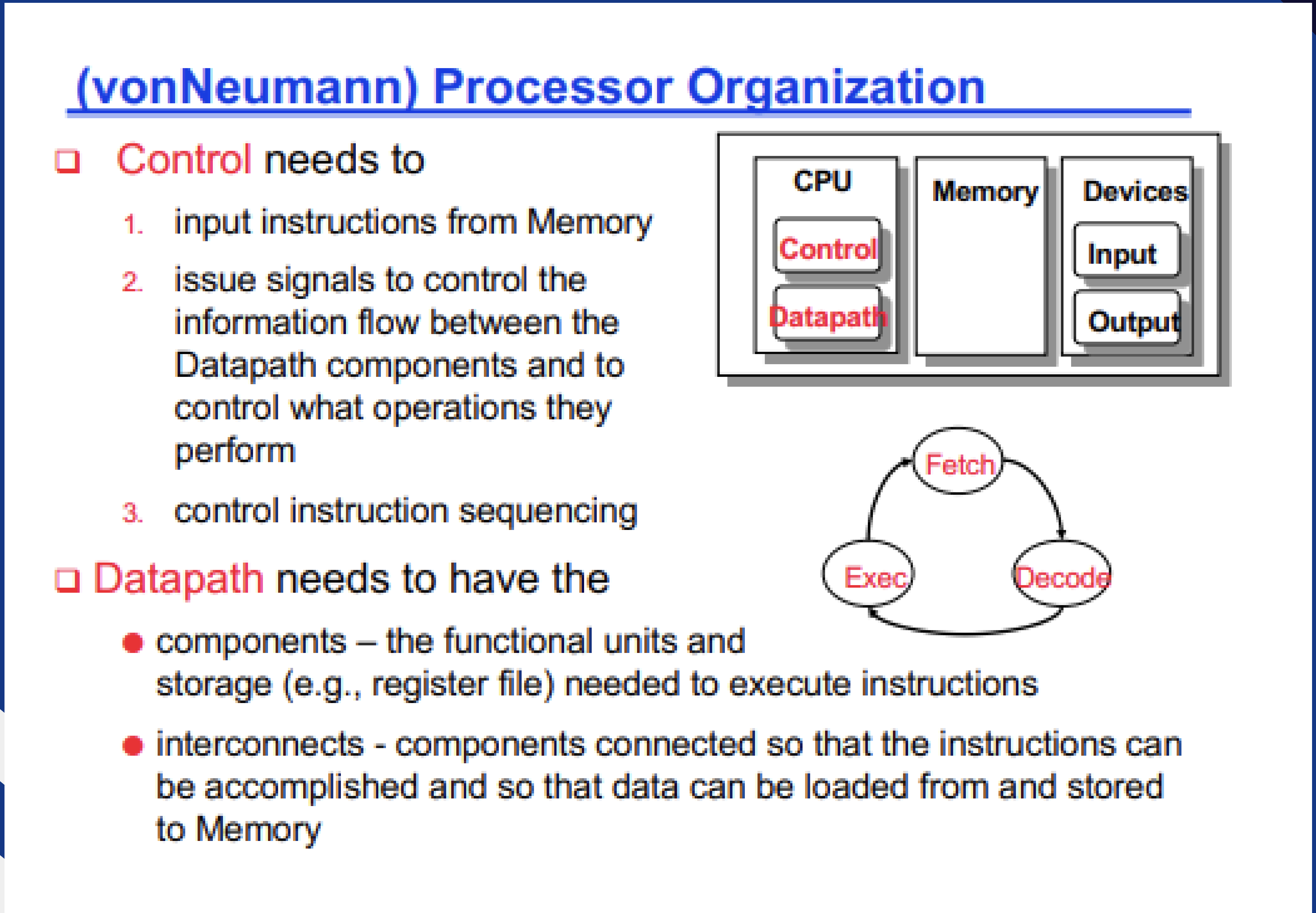
Processor Organization



(vonNeumann) Processor Org

- Control needs to
  1. input instructions from Memory
  2. issue signals to control the information flow between the Datapath components and to control what operations they perform
  3. control instruction sequencing
- Datapath needs to have the
  - components – the functional units and storage (e.g., register file) needed to execute instructions
  - interconnects - components connected so that the instructions can be accomplished and so that data can be loaded from and stored to Memory

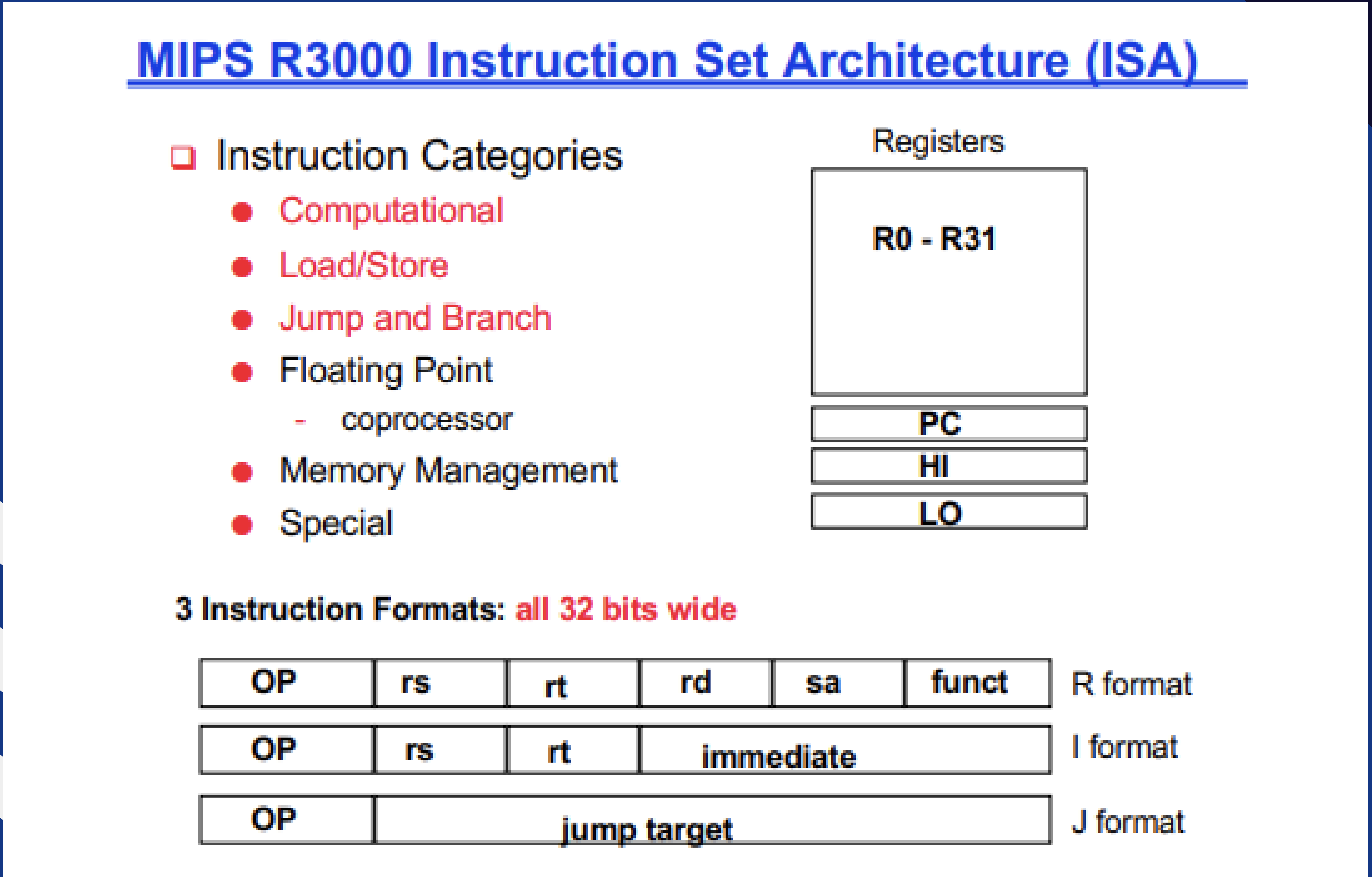
Processor Organization



(vonNeumann) Processor Organization

- ❑ Control needs to
  1. input instructions from Memory
  2. issue signals to control the information flow between the Datapath components and to control what operations they perform
  3. control instruction sequencing
- ❑ Datapath needs to have the
  - ● components – the functional units and storage (e.g., register file) needed to execute instructions
  - ● interconnects - components connected so that the instructions can be accomplished and so that data can be loaded from and stored to Memory

MIPS Arithmetic Instructions



MIPS R3000 Instruction Set Architecture (ISA)

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

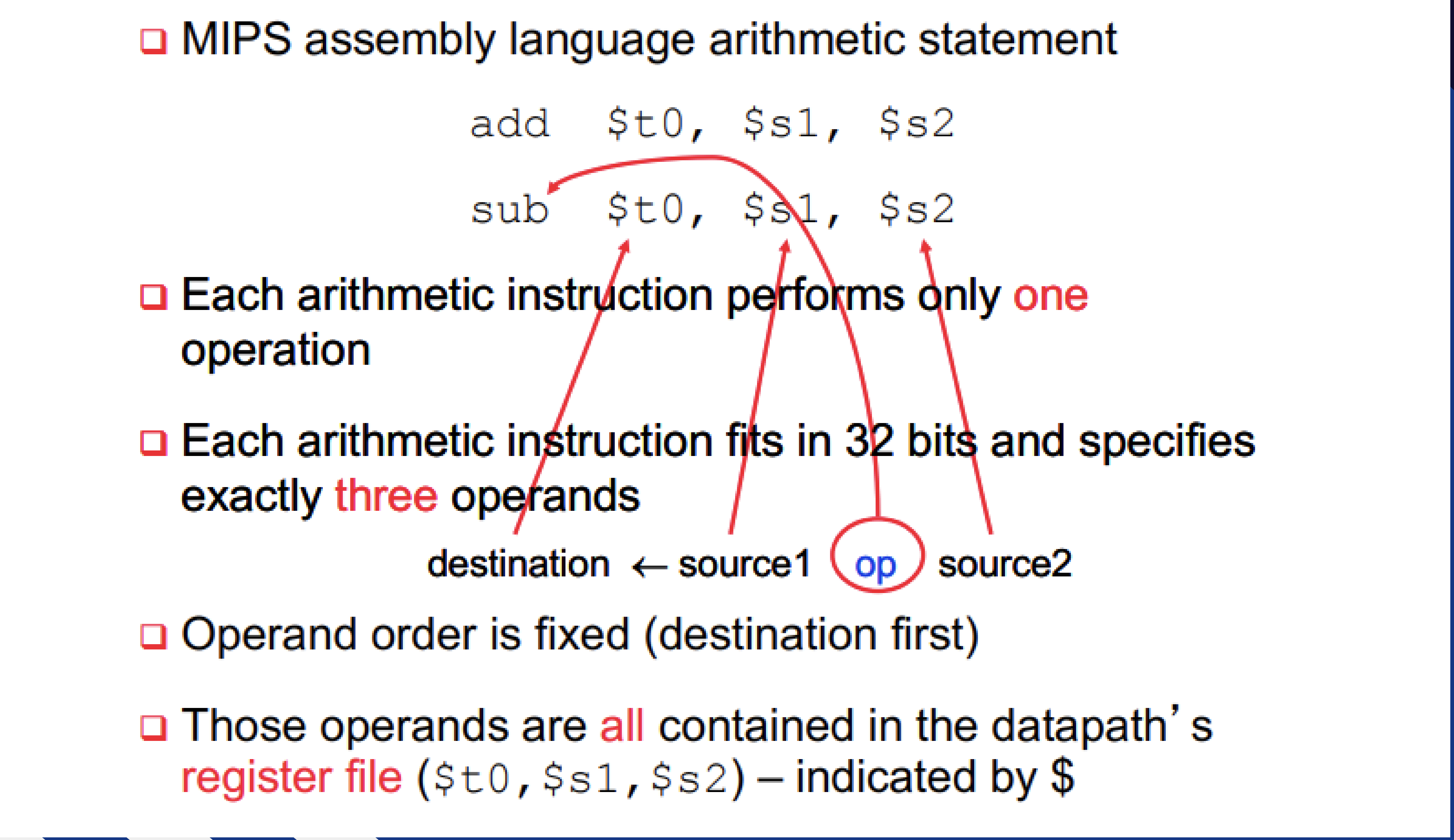
Registers

R0 - R31
PC
HI
LO

3 Instruction Formats: all 32 bits wide

OP	rs	rt	rd	sa	funct	R format
OP	rs	rt	immediate			I format
OP	jump target					J format

MIPS Arithmetic Instructions



- MIPS assembly language arithmetic statement

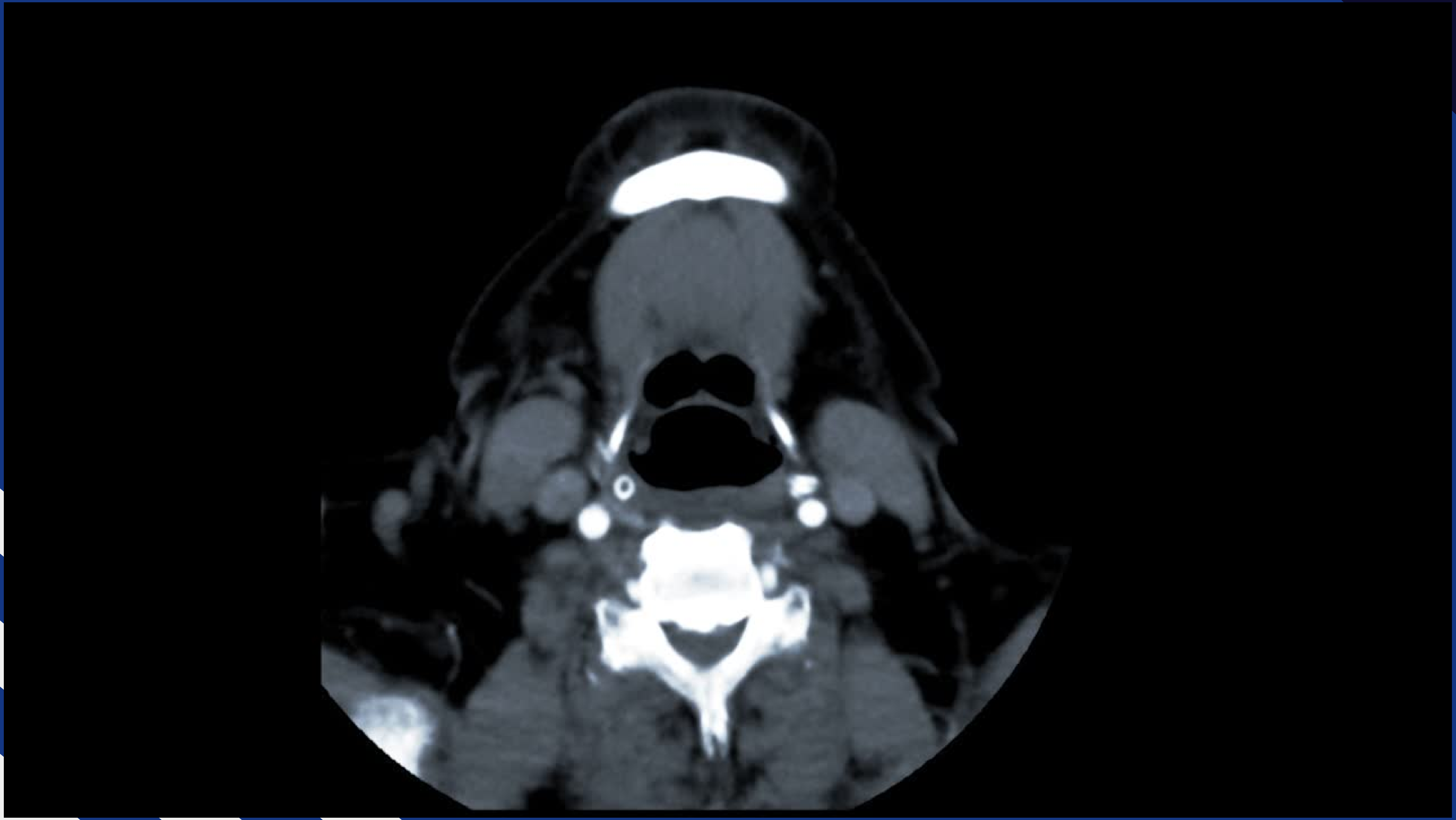
```
add  $t0, $s1, $s2
sub  $t0, $s1, $s2
```

- Each arithmetic instruction performs only one operation
- Each arithmetic instruction fits in 32 bits and specifies exactly three operands

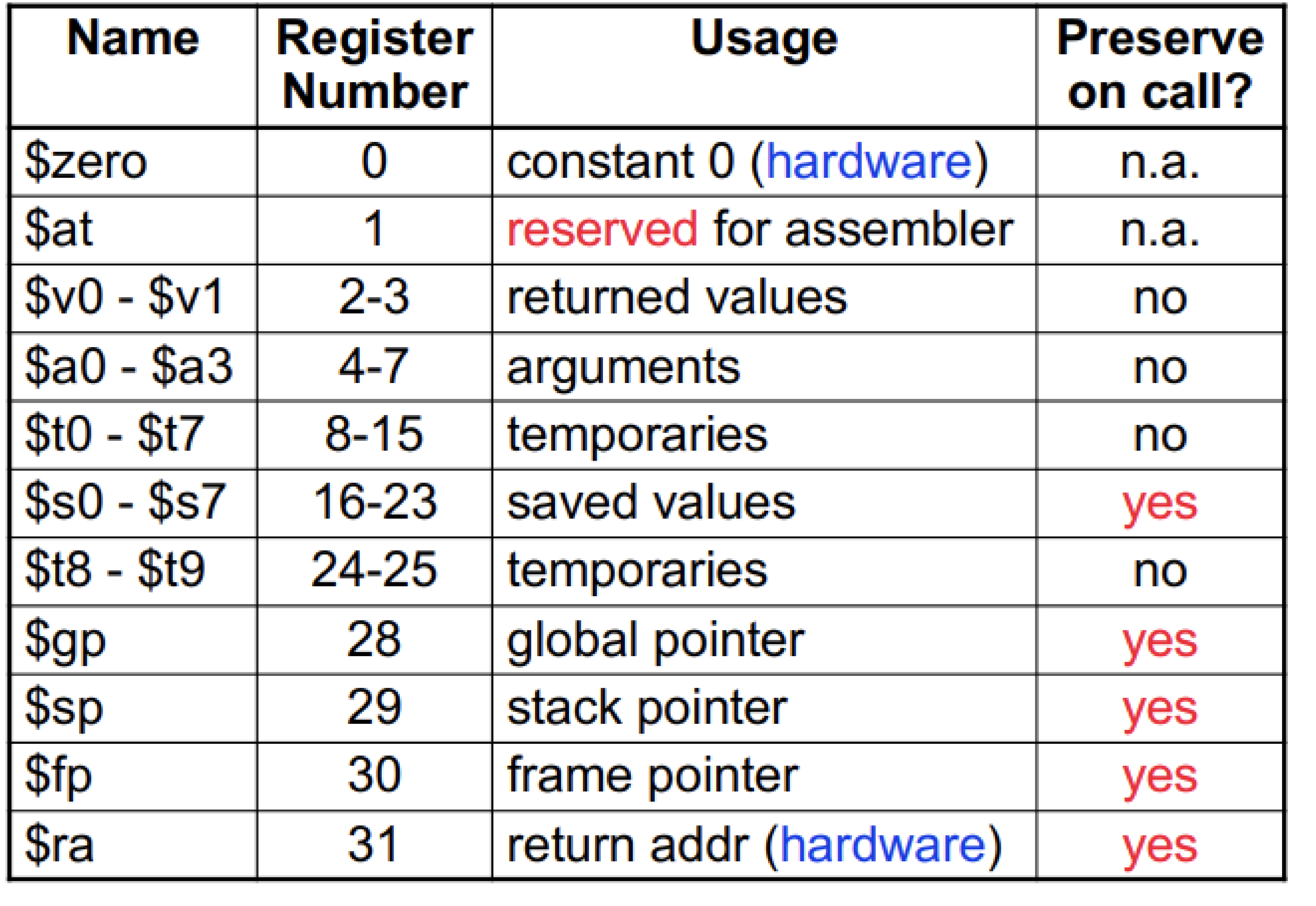
destination ← source1 op source2

- Operand order is fixed (destination first)
- Those operands are all contained in the datapath's register file (`$t0,$s1,$s2`) – indicated by $

MIPS Arithmetic Instructions

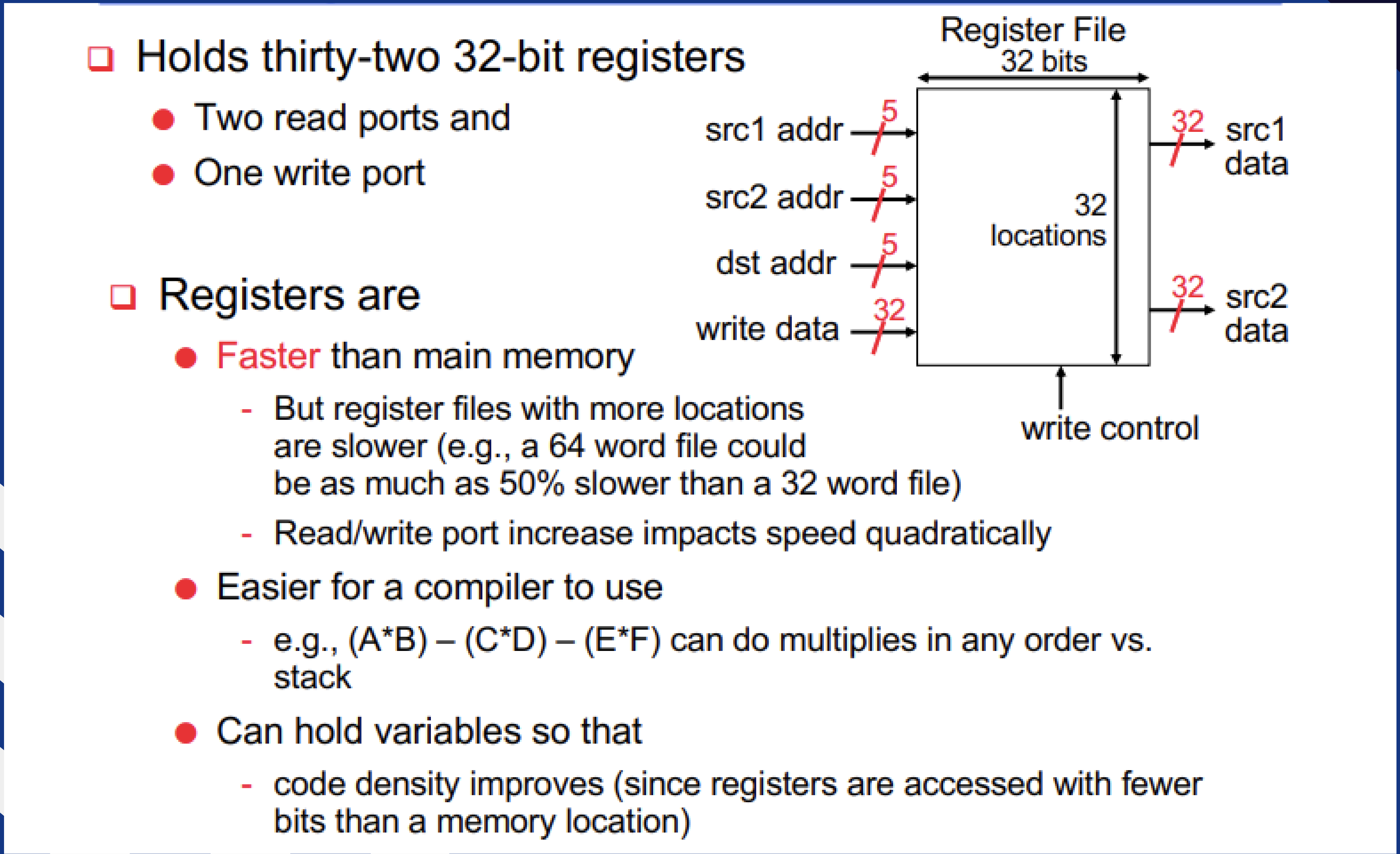


MIPS Registeration Convention



Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	no
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

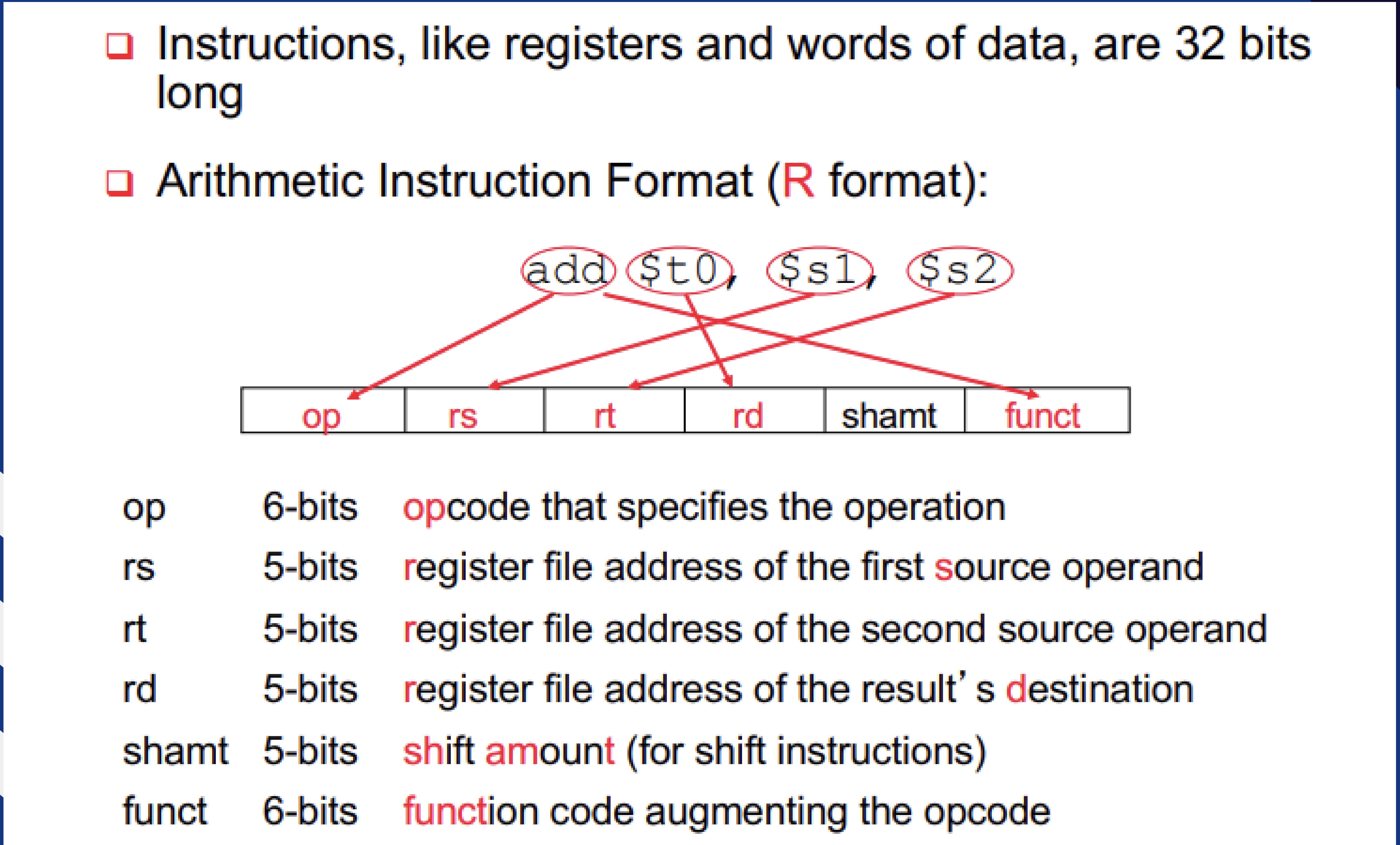
MIPS Register File



- Holds thirty-two 32-bit registers
  - Two read ports and
  - One write port

- Registers are
  - Faster than main memory
    - But register files with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file)
    - Read/write port increase impacts speed quadratically
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - Can hold variables so that
    - code density improves (since registers are accessed with fewer bits than a memory location)

Machine Language Add Instruction



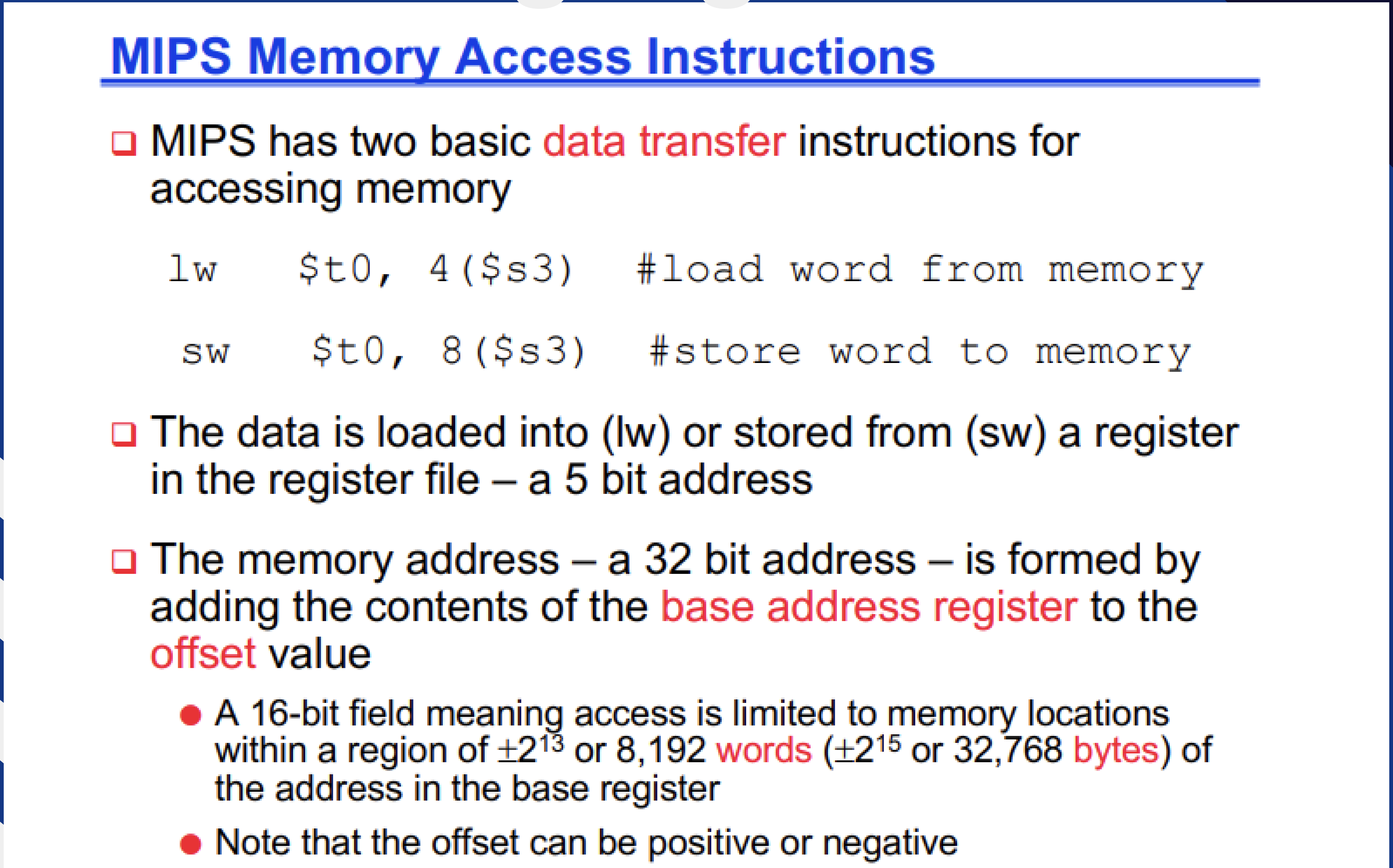
- Instructions, like registers and words of data, are 32 bits long
- Arithmetic Instruction Format (R format):

add $t0, $s1, $s2

op	rs	rt	rd	shamt	funct

op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result’s destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

Machine Language Add Instruction



MIPS Memory Access Instructions

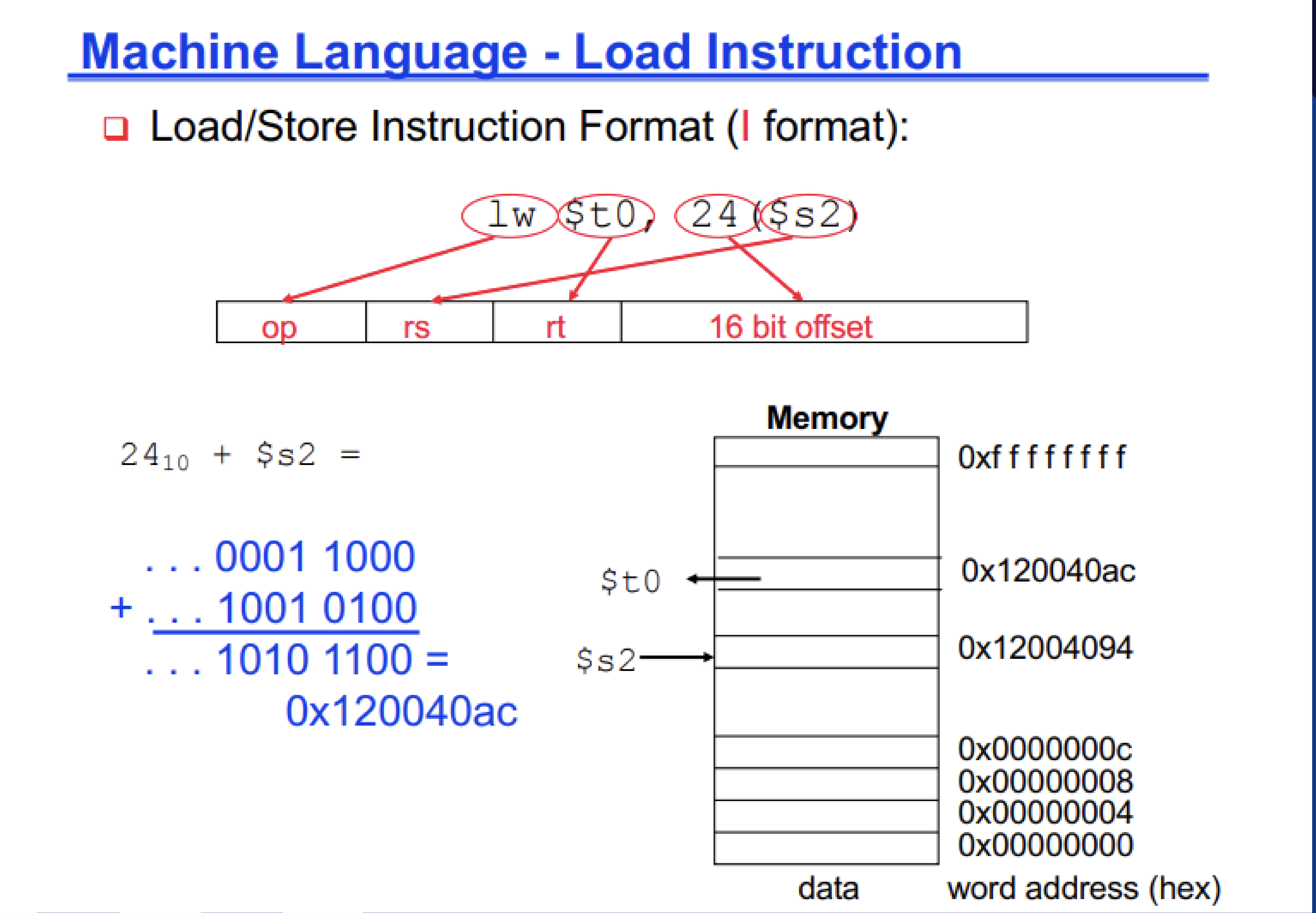
- MIPS has two basic data transfer instructions for accessing memory

```
 lw   $t0, 4($s3)  #load word from memory

 sw   $t0, 8($s3)  #store word to memory
```

- The data is loaded into (lw) or stored from (sw) a register in the register file – a 5 bit address
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value
  - A 16-bit field meaning access is limited to memory locations within a region of $\pm 2^{13}$ or 8,192 words ($\pm 2^{15}$ or 32,768 bytes) of the address in the base register
  - Note that the offset can be positive or negative

Machine Language Add Instruction



Machine Language - Load Instruction

- Load/Store Instruction Format (I format):

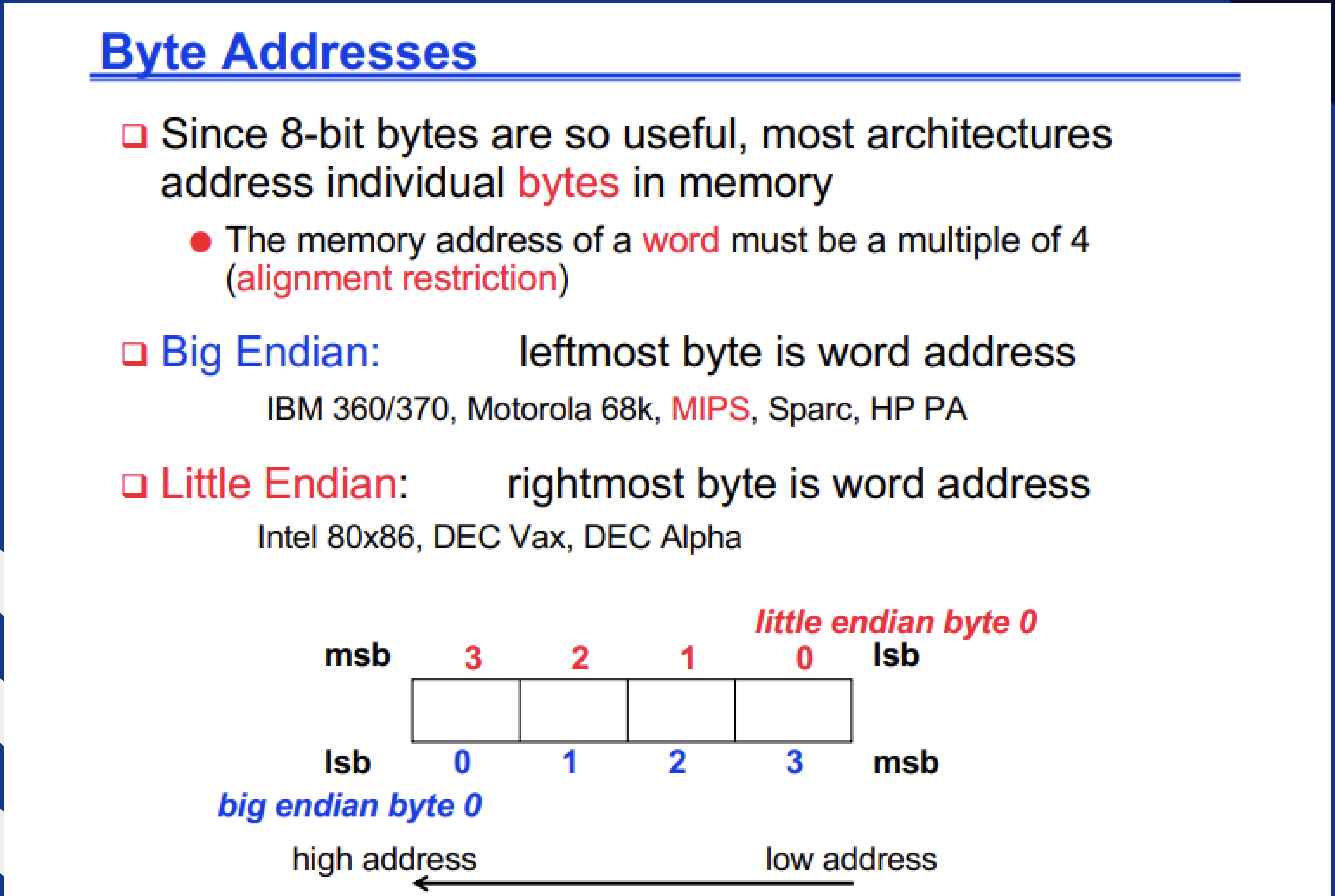
```
lw $t0, 24($s2)
```

op	rs	rt	16 bit offset

24_{10} + \$s2 =

```
  . . . 0001 1000
+ . . . 1001 0100
  . . . 1010 1100 =
        0x120040ac
```

Byte Addresses



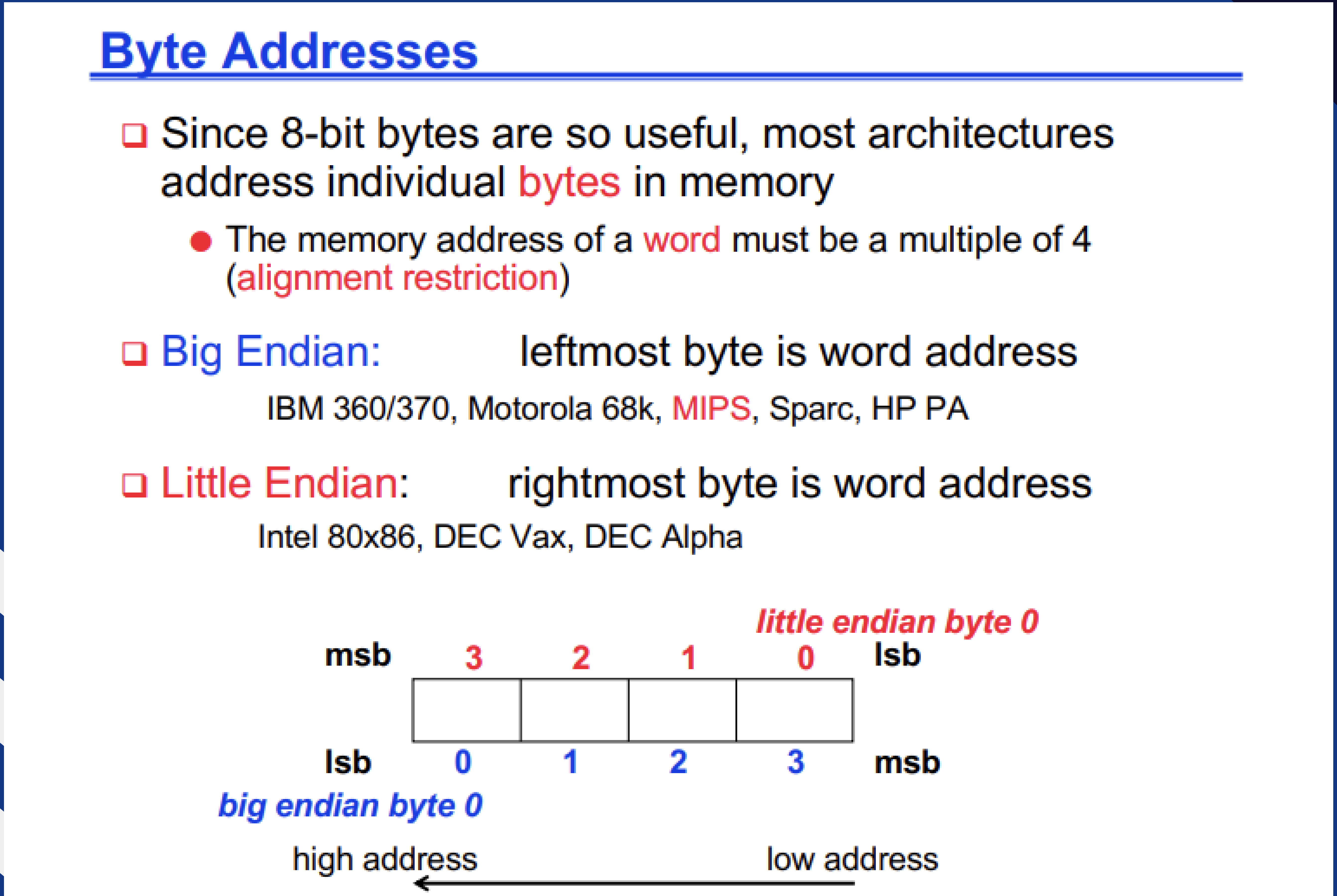
Byte Addresses

- ❑ Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - The memory address of a word must be a multiple of 4 (alignment restriction)
- ❑ Big Endian: leftmost byte is word address

  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- ❑ Little Endian: rightmost byte is word address

  Intel 80x86, DEC Vax, DEC Alpha

Byte Addresses



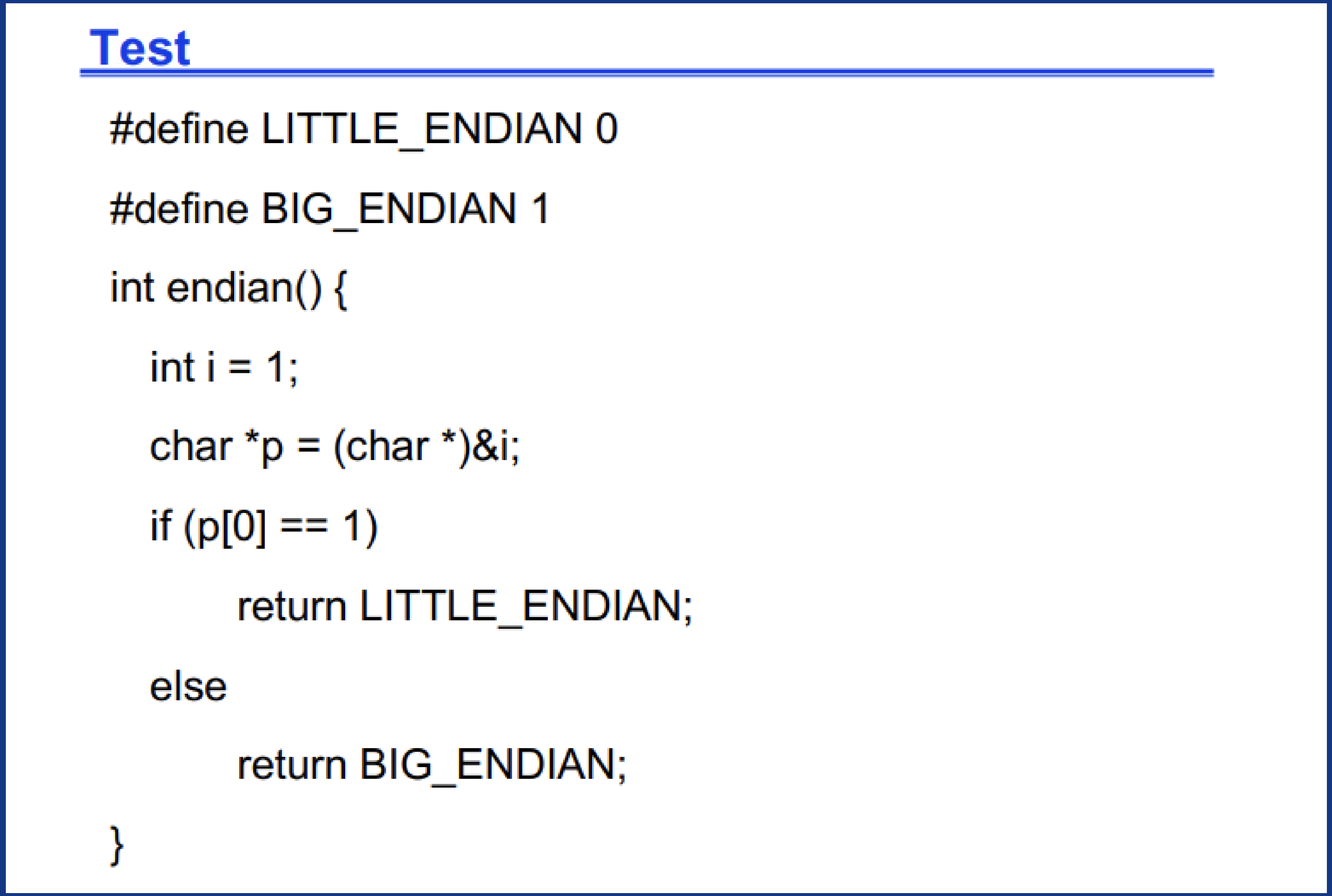
Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - The memory address of a word must be a multiple of 4 (alignment restriction)
- Big Endian: leftmost byte is word address

  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: rightmost byte is word address

  Intel 80x86, DEC Vax, DEC Alpha

Byte Addresses



Test

```
#define LITTLE_ENDIAN 0
#define BIG_ENDIAN 1
int endian() {
   int i = 1;
   char *p = (char *)&i;
   if (p[0] == 1)
         return LITTLE_ENDIAN;
   else
         return BIG_ENDIAN;
}
```