Principles of Computers
8th Lecture

http://d3s.mff.cuni.cz/~jezek

Department of Distributed and Dependable Systems

Pavel Ježek, Ph.D.

pavel.jezek@d3s.mff.cuni.cz

CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

PREVIOUSLY ON

OF

Basic Instructions (6502 vs. x86)

6502 machine code	Intel x86 (IA-32) machine code	Comment
0 **$EA** PC := PC + 1 *6502 assembler:* **NOP**	**0** **$90** EIP := EIP + 1 *Intel assembler:* **NOP**	← Offset from instruction's start (base) address ← Actual bytes of instruction's machine code *No operation (just do nothing and continue to next instruction)*
0 1 2 **$4C** xx_0 xx_1 PC := $\$xx_1xx_0$ *6502 assembler:* **JMP $**$xx_1xx_0$	**0 1 2 3 4** **$E9** xx_0 xx_1 xx_2 xx_3 EIP := $\$xx_3xx_2xx_1xx_0$ *Intel assembler:* **JMP** $xx_3xx_2xx_1xx_0$**h**	← Offset from instruction's start (base) address ← Actual bytes of instruction's machine code *Direct jump to target address x*

JMP 00000005h in assembler is 15 bytes in UTF-8 encoding:

```
00000000: 4A 4D 50 20 30 30 30 30|30 30 30 35 68 0D 0A      |JMP 00000005h..
```

x86 assembler (compiler)

In machine code = 5 bytes:

E9 05 00 00 00

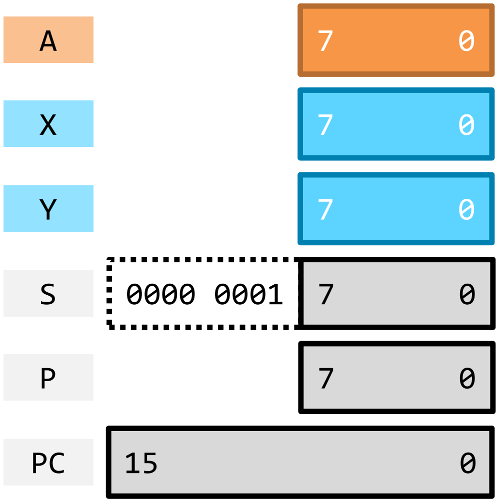
Department of Distributed and Dependable Systems

Typical ISA Arithmetic Instructions

MIPS: a := b *op* c

x86, 6502: a := a *op* b

6502 Registers (Accumulator Architecture)



6502: **8-bit CPU** with **16-bit** logical and physical **address space**s (1:1 mapping between logical and physical addresses, i.e. logical address = physical address)

Move (Transfer) Value Between Registers

`LDA #$xx`	`A := xx`
`LDA $xxxx`	`A := ($xxxx)^`
`LDX imm/addr`	`X := imm/addr`
`LDY imm/addr`	`Y := imm/addr`
`STA $xxxx`	`($xxxx)^ := A`
`STX addr`	`($addr)^ := X`
`STY addr`	`($addr)^ := Y`

`TAX`	`X := A`
`TXA`	`A := X`
`TAY`	`Y := A`
`TYA`	`A := Y`
`TSX`	`X := S`
`TXS`	`S := X`

Setting Flags

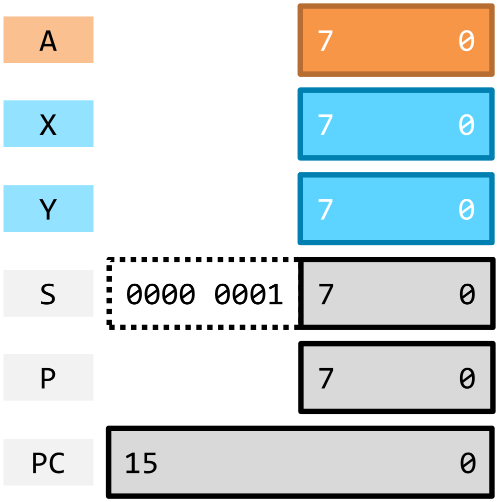
LDA #$xx	A := xx
LDA $xxxx	A := ($xxxx)^
LDX imm/addr	X := imm/addr
LDY imm/addr	Y := imm/addr
STA $xxxx	($xxxx)^ := A
STX addr	($addr)^ := X
STY addr	($addr)^ := Y

```
      7654 3210
P     N... ..Z.
```

```
P.Negative := target.7

if target = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

TAX	X := A
TXA	A := X
TAY	Y := A
TYA	A := Y
TSX	X := S
TXS	S := X

6502 Registers (Accumulator Architecture)



6502: **8-bit CPU** with **16-bit** logical and physical **address space**s (1:1 mapping between logical and physical addresses, i.e. logical address = physical address)

Setting Flags

LDA #$xx	A := xx
LDA $xxxx	A := ($xxxx)^
LDX imm/addr	X := imm/addr
LDY imm/addr	Y := imm/addr
STA $xxxx	($xxxx)^ := A
STX addr	($addr)^ := X
STY addr	($addr)^ := Y

TAX	X := A
TXA	A := X
TAY	Y := A
TYA	A := Y
TSX	X := S
TXS	S := X

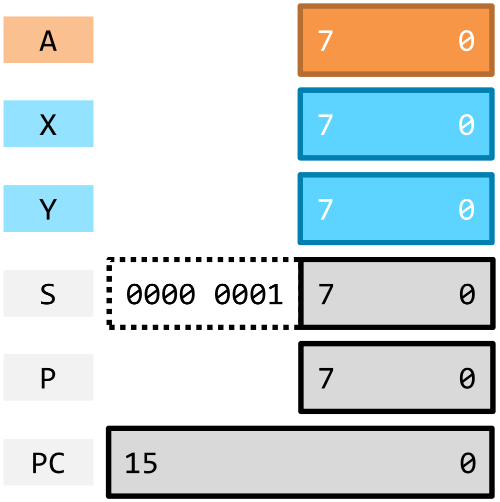
	7654 3210
P	N... ..Z.

```
P.Negative := target.7

if target = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

CLC	P.Carry := 0
SEC	P.Carry := 1

Department of Distributed and Dependable Systems

6502 Registers (Accumulator Architecture)



6502: **8-bit CPU** with **16-bit** logical and physical **address space**s (1:1 mapping between logical and physical addresses, i.e. logical address = physical address)

Bitwise Operations

```
ORA imm/addr      A := A BitwiseOr imm/addr

AND imm/addr      A := A BitwiseAnd imm/addr

EOR imm/addr      A := A BitwiseXor imm/addr

? NOT             EOR #$FF

ASL A             A := A shl 1

LSR A             A := A shr 1

ROL A             A := A rol 1

ROR A             A := A ror 1
```

```
P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

Oring 16-bit Numbers (e.g. Little Endian)

MSB of A stored at $A001								LSB of A stored at $A000							
A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0

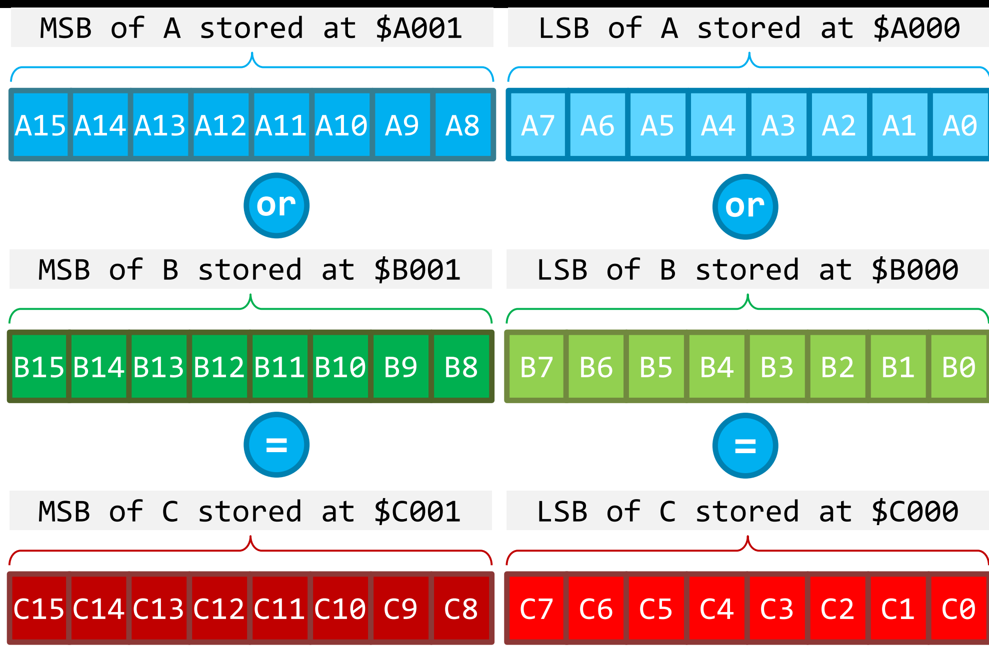
or

MSB of B stored at $B001								LSB of B stored at $B000							
B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

=

MSB of C stored at $C001								LSB of C stored at $C000							
C15	C14	C13	C12	C11	C10	C9	C8	C7	C6	C5	C4	C3	C2	C1	C0

Oring 16-bit Numbers (e.g. Little Endian)




```
LDA $A000
ORA $B000
STA $C000

LDA $A001
ORA $B001
STA $C001
```

Adding Binary Numbers

• • •	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit index
• • •	1	0	0	1	0	1	1	1	0	1	0	0	0	1	1	0	1	0	1	1	Input A
+																					
• • •	0	1	1	1	0	1	1	1	0	1	0	1	0	0	0	0	1	1	1	0	Input B
=																					
• • •	0	0	0	0	1	1	1	0	1	0	0	1	0	1	1	1	1	0	0	1	Result

Adding Binary Numbers

• • •	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit index
• • •	1	0	0	1	0	1	1	1	0	1	0	0	0	1	1	0	1	0	1	1	Input A
• • •	0	1	1	1	0	1	1	1	0	1	0	1	0	0	0	0	1	1	1	0	Input B
																				0	Carry
• • •	0	0	0	0	1	1	1	0	1	0	0	1	0	1	1	1	1	0	0	1	Result

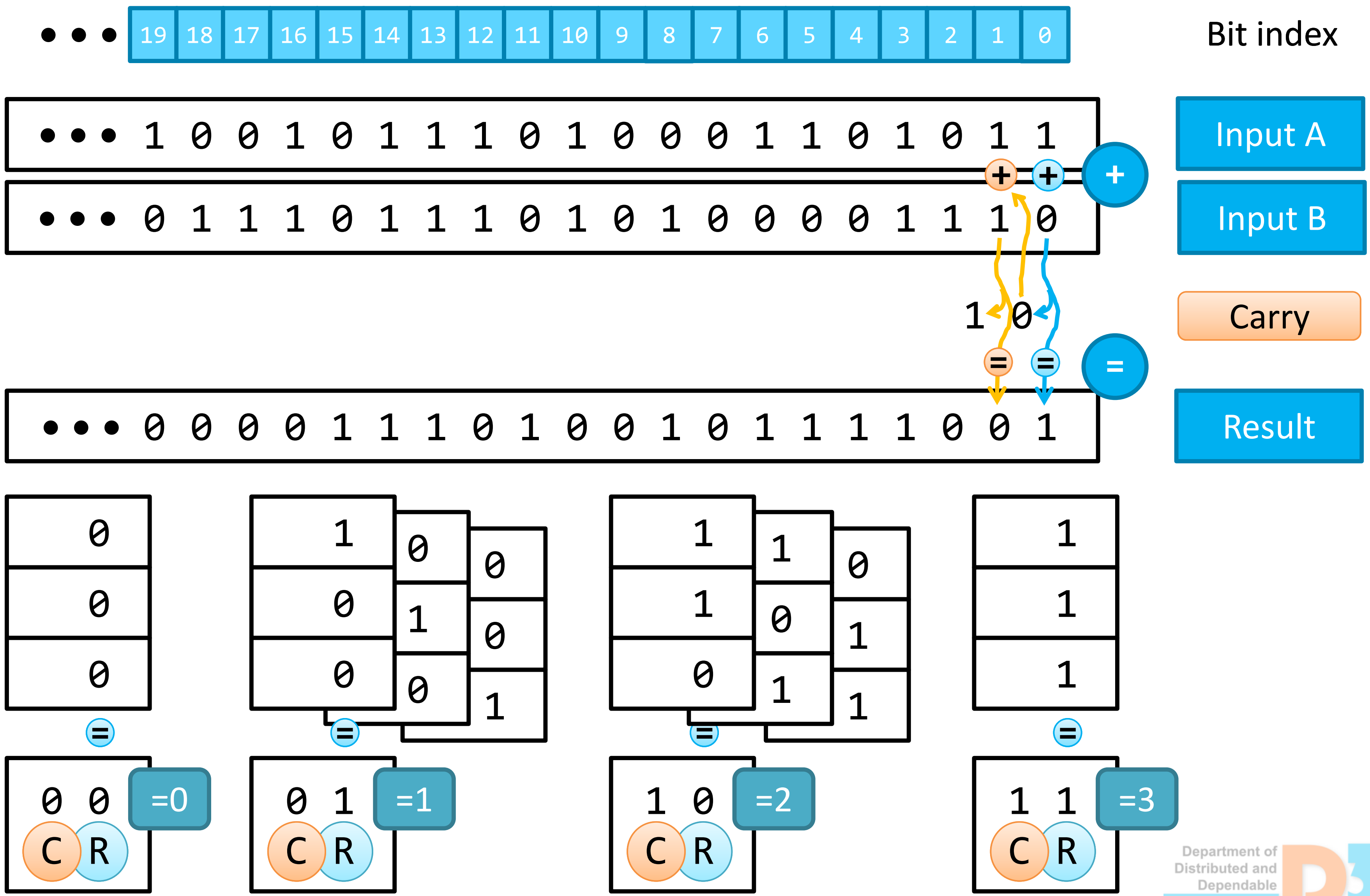
+

=

Adding Binary Numbers

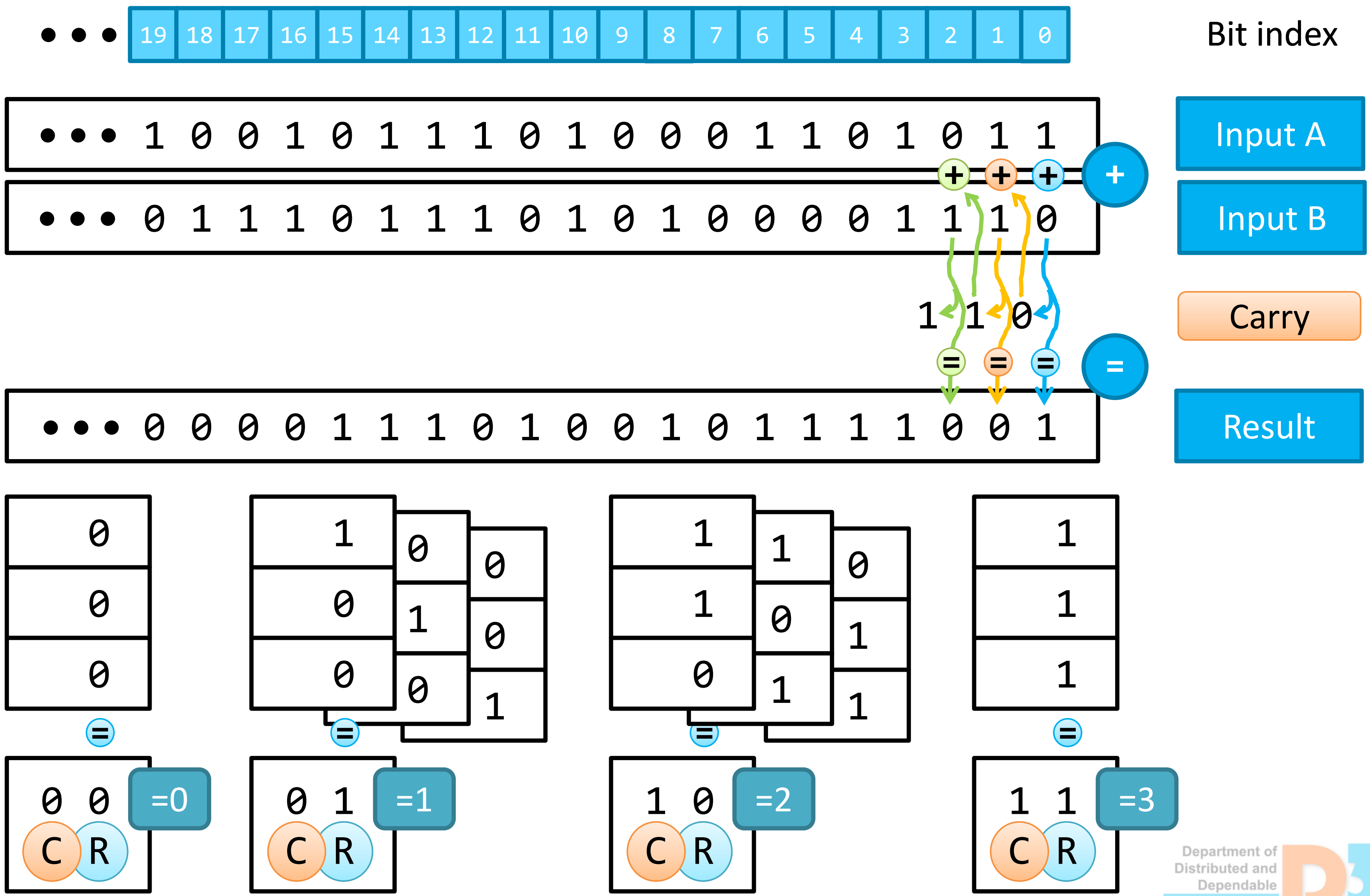
		19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Bit index	• • •	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Input A	• • •	1	0	0	1	0	1	1	1	0	1	0	0	0	1	1	0	1	0	1	1	+
Input B	• • •	0	1	1	1	0	1	1	1	0	1	0	1	0	0	0	0	1	1	1	0	
Carry																				1	0	=
Result	• • •	0	0	0	0	1	1	1	0	1	0	0	1	0	1	1	1	1	0	0	1	

Adding Binary Numbers



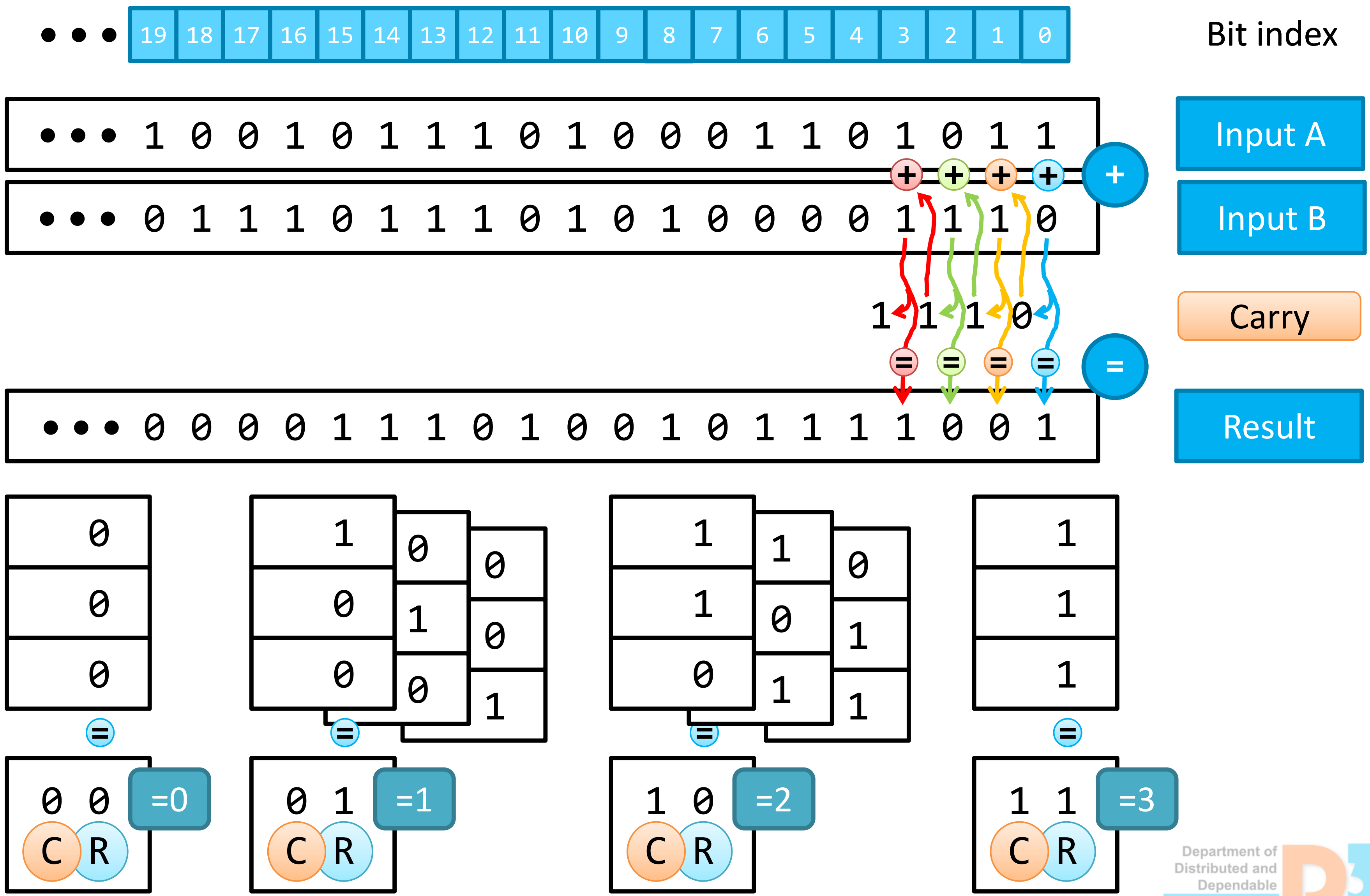
Department of Distributed and Dependable Systems

Adding Binary Numbers

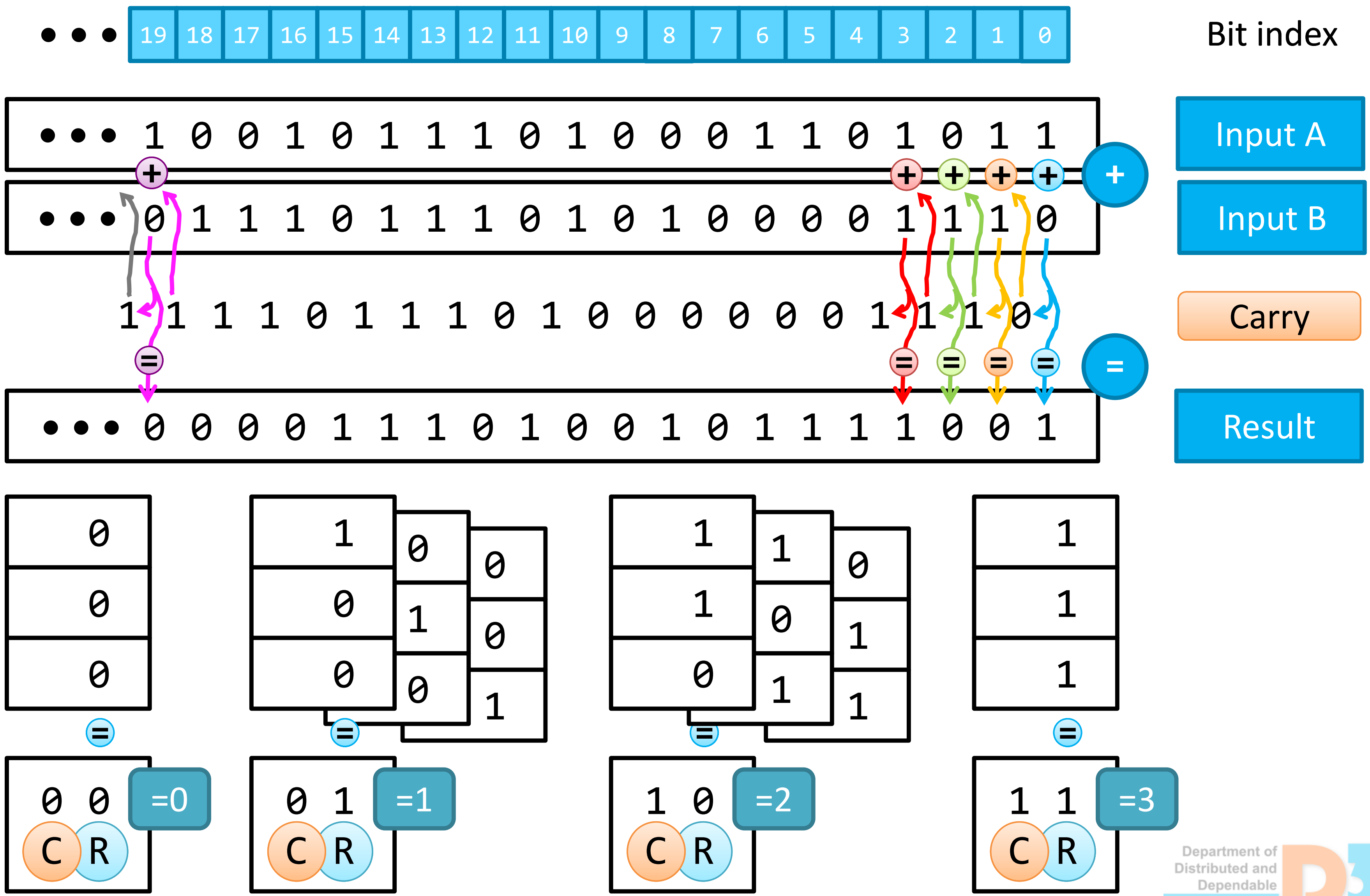


Department of Distributed and Dependable Systems

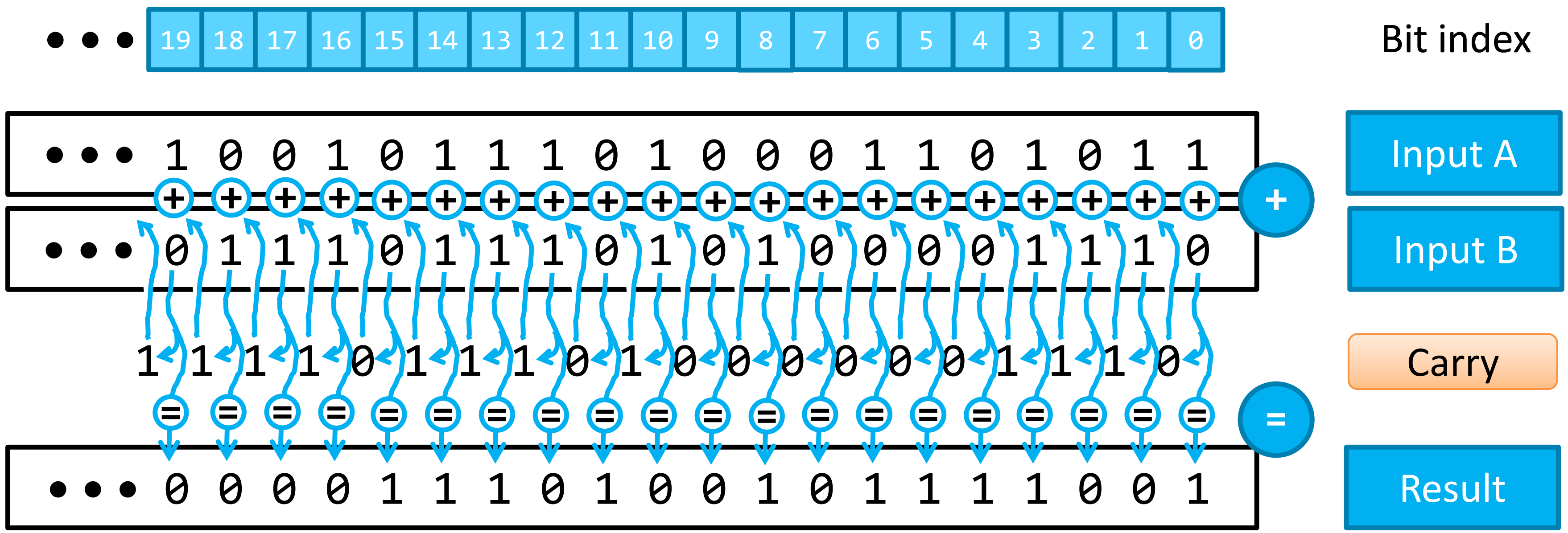
Adding Binary Numbers



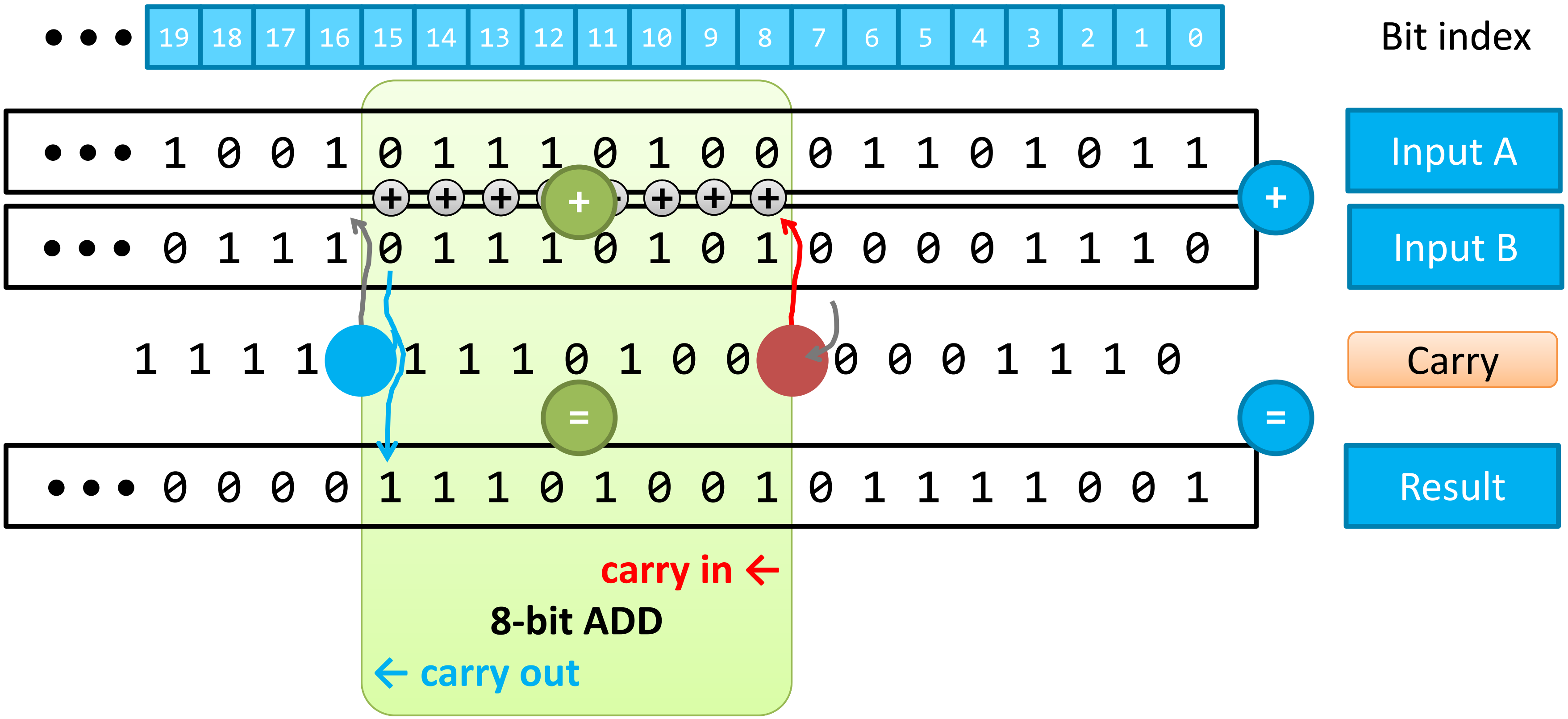
Adding Binary Numbers



Adding Binary Numbers (1-bit addition = 1 unit of work)



Adding Binary Numbers (8-bit addition = 1 unit of work)



Integer Operations

carry in

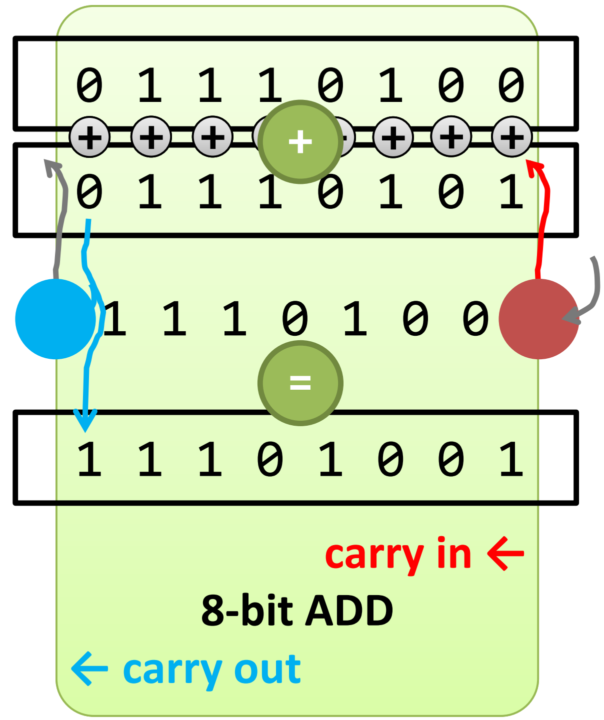
```
ADC imm/addr
```

carry out

```
result := A + imm/addr + P.Carry
P.Carry := result.8
A := result.7 … result.0
```

```
P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

	7654 3210
P	N... ..ZC



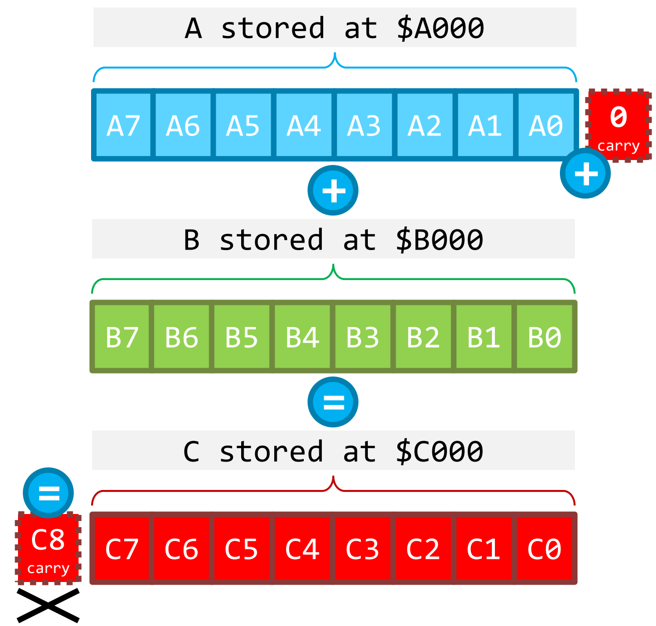
Integer Operations (Adding 8-bit Numbers)

```
ADC imm/addr
```

```
result := A + imm/addr + P.Carry
P.Carry := result.8
A := result.7 … result.0
```

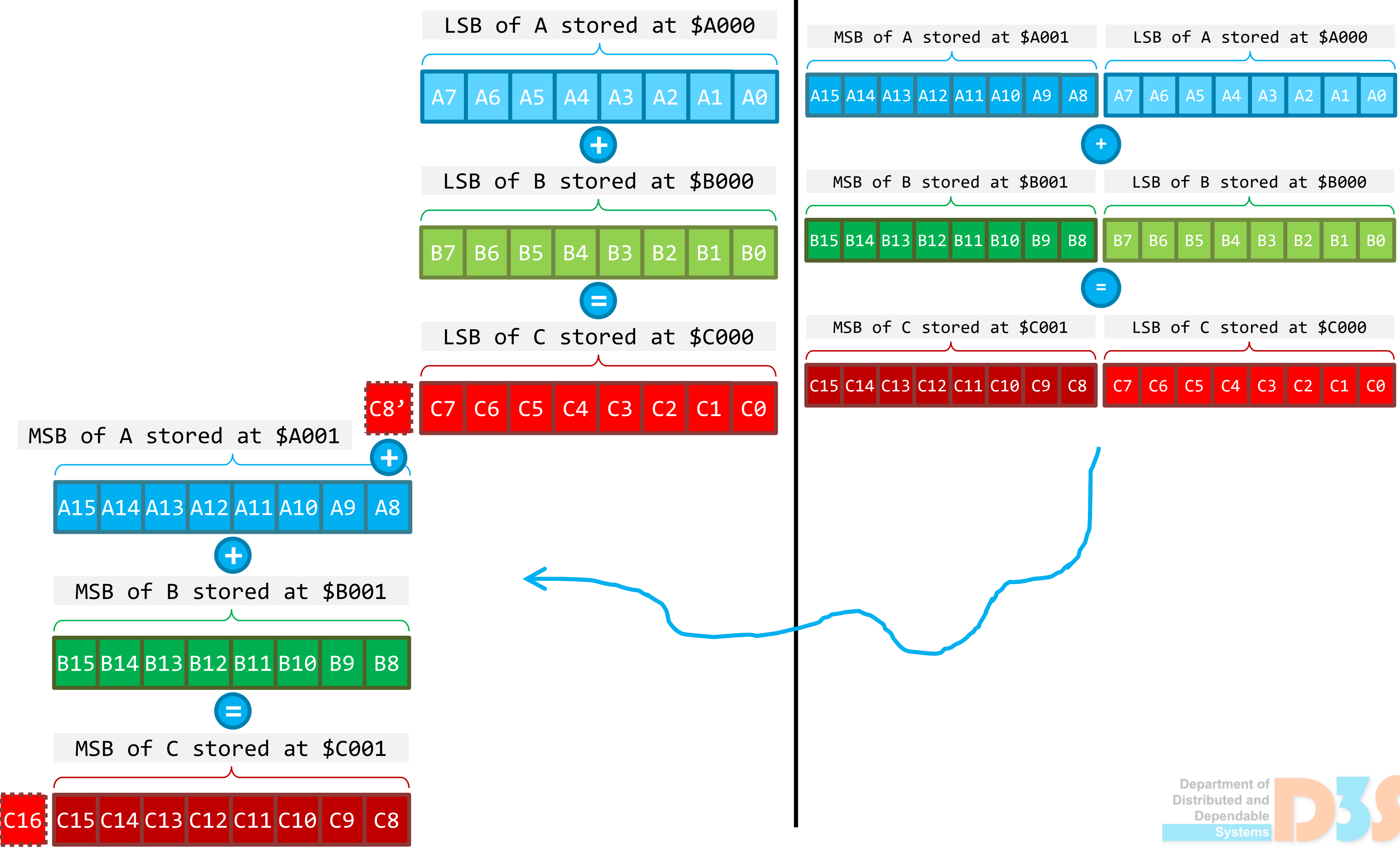
```
P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

P — 7654 3210 — N... ..ZC

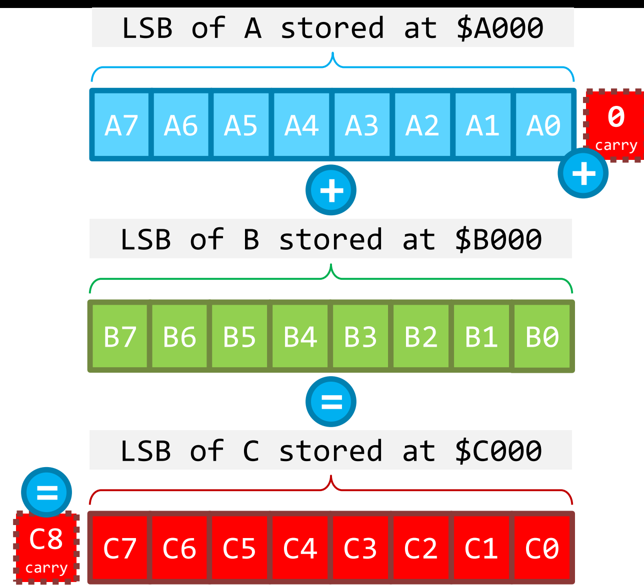



```
LDA $A000
CLC
ADC $B000
STA $C000
```

Adding 16-bit Numbers (e.g. Little Endian)



Adding 16-bit Numbers (e.g. Little Endian)

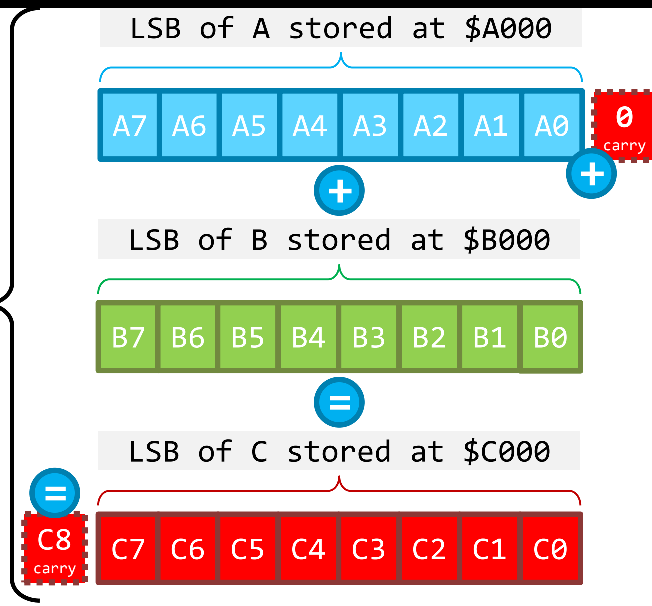



```
ADC imm/addr
```

```
result := A + imm/addr + P.Carry
P.Carry := result.8
A := result.7 … result.0

P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

Adding 16-bit Numbers (e.g. Little Endian)

```
LDA $A000
CLC
ADC $B000
STA $C000
```



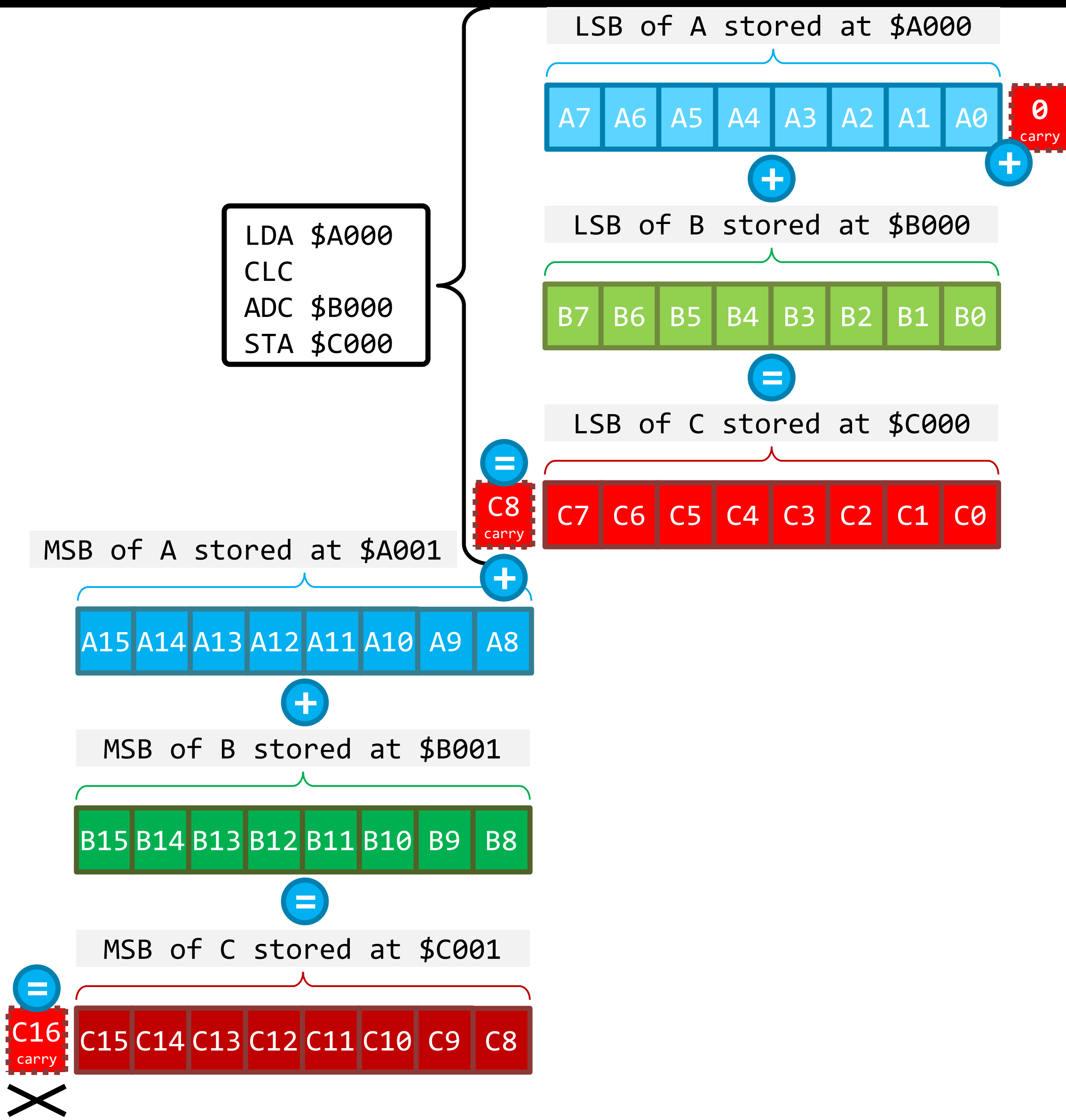

```
ADC imm/addr
```

```
result := A + imm/addr + P.Carry
P.Carry := result.8
A := result.7 … result.0

P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

Department of Distributed and Dependable Systems

Adding 16-bit Numbers (e.g. Little Endian)

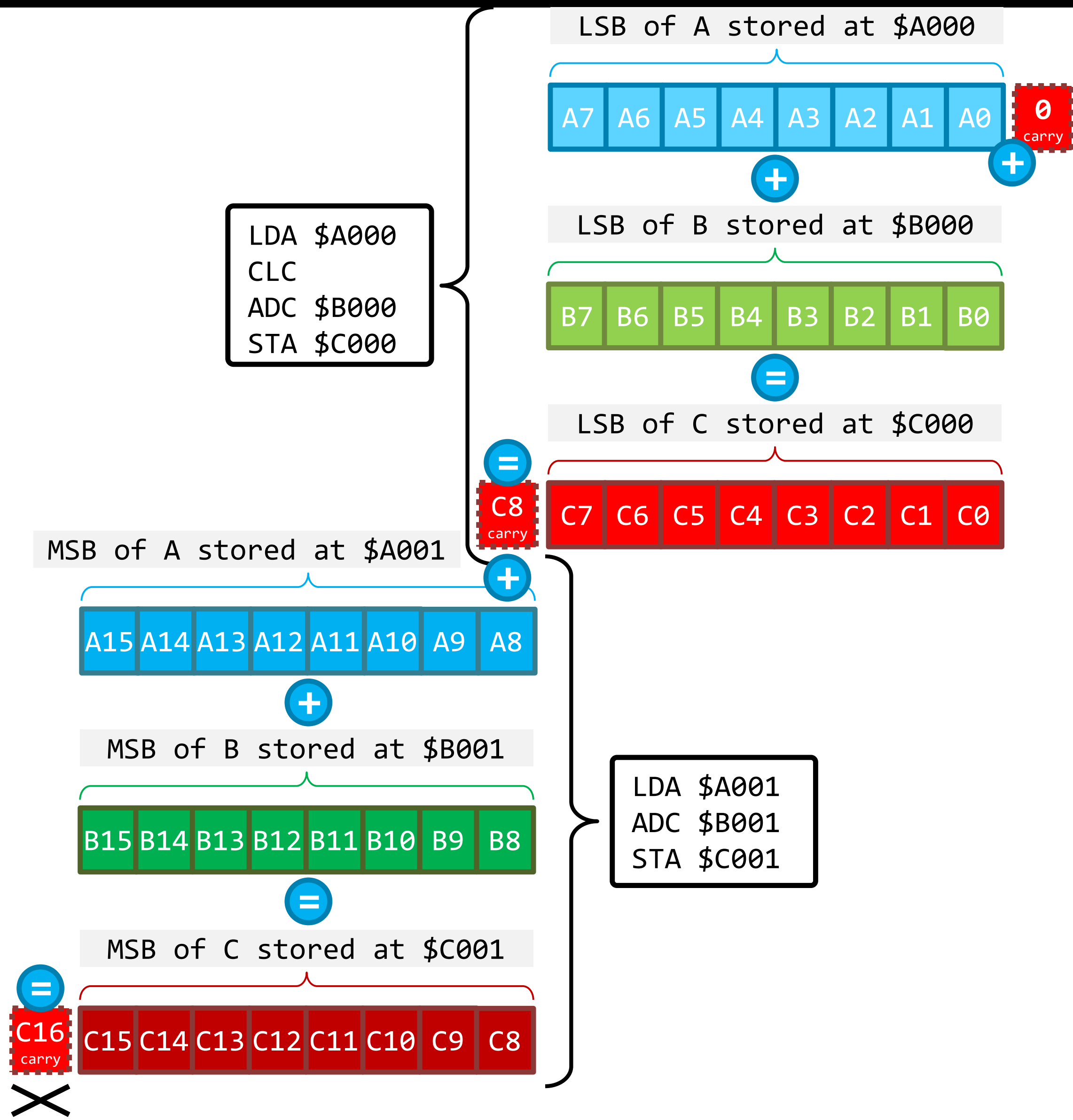
```
LDA $A000
CLC
ADC $B000
STA $C000
```




```
ADC imm/addr
```

```
result := A + imm/addr + P.Carry
P.Carry := result.8
A := result.7 … result.0

P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

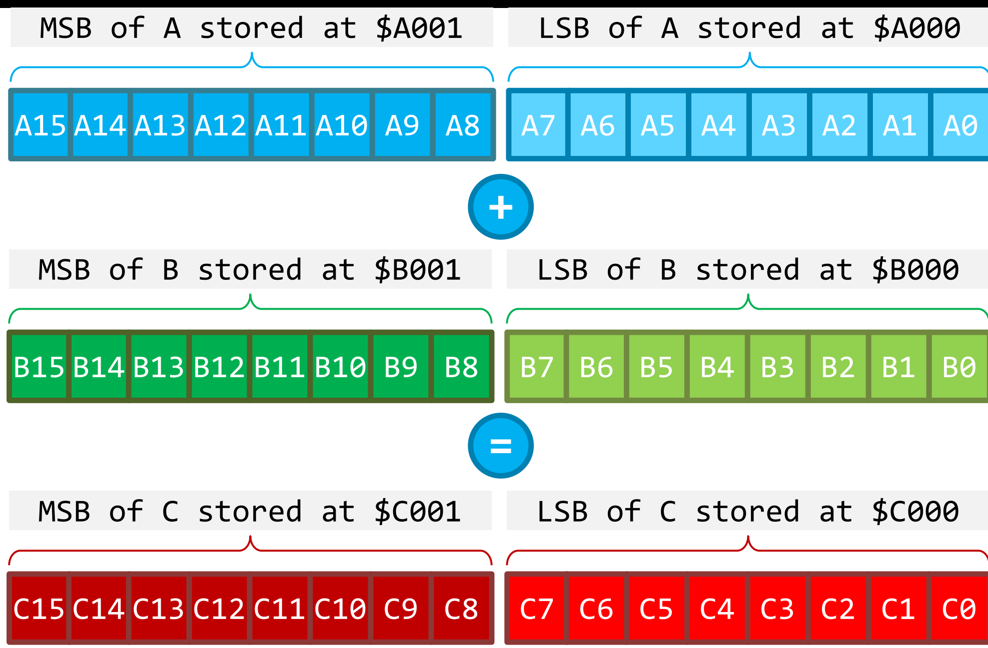
Adding 16-bit Numbers (e.g. Little Endian)




```
ADC imm/addr
```

```
result  := A + imm/addr + P.Carry
P.Carry := result.8
A := result.7 … result.0

P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

Adding 16-bit Numbers (e.g. Little Endian)




```
LDA $A000
CLC
ADC $B000
STA $C000
LDA $A001
ADC $B001
STA $C001
```

ADC imm/addr

```
result := A + imm/addr + P.Carry
P.Carry := result.8
A := result.7 … result.0

P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

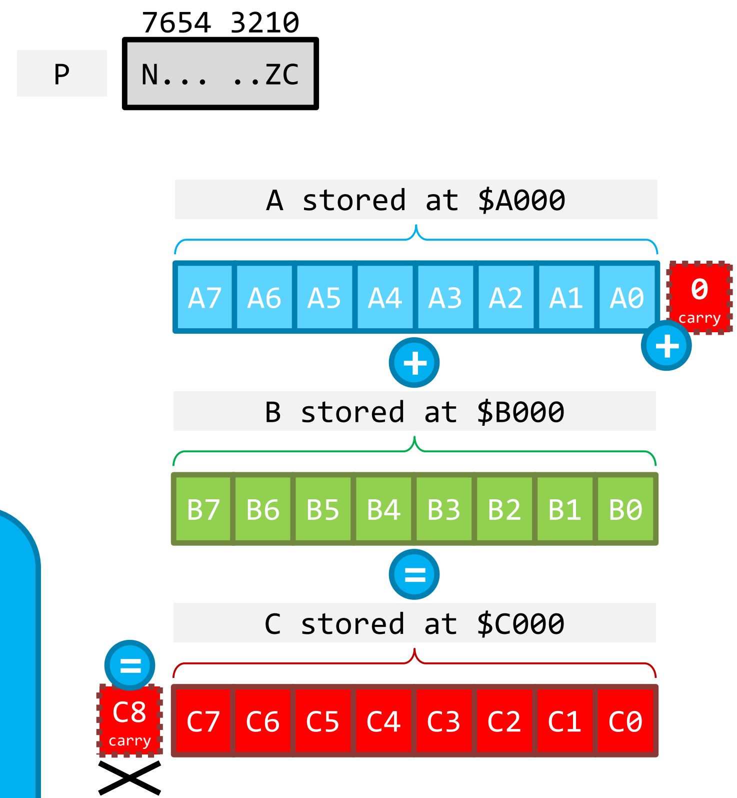
Integer Operations (Adding 8-bit Numbers)

```
ADC imm/addr
```

```
result := A + imm/addr + P.Carry
P.Carry := result.8
A := result.7 … result.0
```

```
P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

Signed Integers?




```
LDA $A000
CLC
ADC $B000
STA $C000
```

Integer Operations – Increments and Decrements (only X and Y)

`ADC imm/addr`	`result := A + imm/addr + P.Carry`
	`P.Carry := result.8`
	`A := result.7 … result.0`

INX
INY
DEX
DEY
	`X := X + 1`
	`Y := Y + 1`
	`X := X - 1`
	`Y := Y - 1`

```
P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

```
P.Negative := X/Y.7

if X/Y = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

Integer Operations – Subtraction?

```
ADC  imm/addr
```

```
result := A + imm/addr + P.Carry
P.Carry := result.8
A := result.7 … result.0
```

```
P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

unsigned numbers

A := value – A

```
INX
INY
DEX
DEY
```

```
X := X + 1
Y := Y + 1
X := X - 1
Y := Y - 1
```

```
P.Negative := X/Y.7

if X/Y = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

Department of Distributed and Dependable Systems

Integer Operations – Subtraction?

```
ADC  imm/addr
```

```
result := A + imm/addr + P.Carry
P.Carry := result.8
A := result.7 … result.0
```

```
P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

A := value – A ≡ A := value + (– A)

```
INX
INY
DEX
DEY
```

```
X := X + 1
Y := Y + 1
X := X – 1
Y := Y - 1
```

```
P.Negative := X/Y.7

if X/Y = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

Department of Distributed and Dependable Systems

Integer Operations – Subtraction? Two's Complement?

ADC imm/addr	result := A + imm/addr + P.Carry P.Carry := result.8 A := result.7 … result.0
INX INY DEX DEY	X := X + 1 Y := Y + 1 X := X - 1 Y := Y - 1

```
P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

unsigned numbers

A := value – A = A := value + (– A)

Negative number ↓ signed representation needed

```
P.Negative := X/Y.7

if X/Y = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

Integer Operations – Subtraction? Two's Complement?

```
ADC imm/addr
```

```
result := A + imm/addr + P.Carry
P.Carry := result.8
A := result.7 … result.0
```

```
P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

unsigned numbers

A := value – A = A := value + (– A)

Negative number → signed representation needed

How to represent non-negative number in two's complement?

```
INX
INY
DEX
DEY
```

```
X := X + 1
Y := Y + 1
X := X - 1
Y := Y - 1
```

```
P.Negative := X/Y.7

if X/Y = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

Integer Operations – Subtraction? Via Two's Complement

```
ADC imm/addr
```

```
result := A + imm/addr + P.Carry
P.Carry := result.8
A := result.7 … result.0
```

```
P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

Addition is commutative

A := value – A ≡ A := value + (– A) ≡ A := (– A) + value

```
INX
INY
DEX
DEY
```

```
X := X + 1
Y := Y + 1
X := X – 1
Y := Y - 1
```

```
P.Negative := X/Y.7

if X/Y = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

Integer Operations – Subtraction? Via Two's Complement

```
ADC imm/addr
```

```
result := A + imm/addr + P.Carry
P.Carry := result.8
A := result.7 … result.0
```

```
P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

```
A := value - A
↓
A := (-A) + value
↓
NOT A
INC A
ADD value
```

Not a problem, that everything is in unsigned arithmetic

A := value – A = A := value + (– A)

= A := (– A) + value

```
INX
INY
DEX
DEY
```

```
X := X + 1
Y := Y + 1
X := X - 1
Y := Y - 1
```

```
P.Negative := X/Y.7

if X/Y = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

Department of Distributed and Dependable Systems

Integer Operations – Subtraction? Via Two's Complement

```
ADC imm/addr
```

```
result := A + imm/addr + P.Carry
P.Carry := result.8
A := result.7 … result.0
```

```
P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

```
A := value - A
↓
A := (-A) + value
↓
NOT A
INC A
ADD value
```

```
A := value - A
↓
A := (-A) + value
↓
EOR #$FF
CLC
ADC #1
CLC
ADC value
```

A := value – A ≡ A := value + (– A)

≡ A := (– A) + value

```
INX
INY
DEX
DEY
```

```
X := X + 1
Y := Y + 1
X := X - 1
Y := Y - 1
```

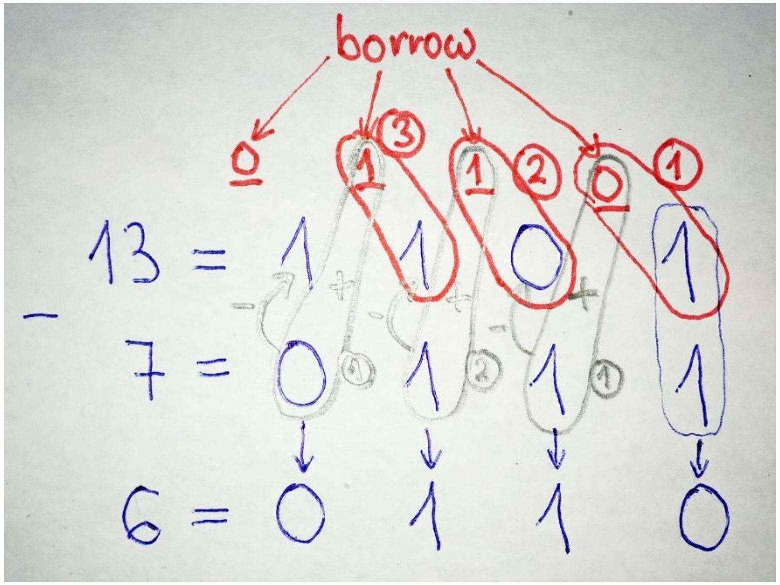
```
P.Negative := X/Y.7

if X/Y = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

Subtract with Borrow (SBB) – e.g. x86 (IA-32)

SBB op1, op2
(N-bit subtract of bits N-1, …, 0)

op1 := op1 – (op2 + CarryFlag)
CarryFlag := Borrow from bit N (not in result) into bit N-1

CarryFlag = 1 then need for Borrow
CarryFlag = 0 then no Borrow



Department of Distributed and Dependable Systems

Integer Operations – Subtraction?

Subtract with Carry (= Not Borrow)

`ADC imm/addr`	`result := A + imm/addr + P.Carry`
	`P.Carry := result.8`
	`A := result.7 … result.0`
	
	`P.Negative := A.7`
	
	`if A = 0 then`
	`  P.Zero := 1`
	`else`
	`  P.Zero := 0;`

`SBC imm/addr`	`result :=` **`A – imm/addr –`** `NOT(P.Carry)` **`P.Carry := NOT(result.7)`** **`A := result.7 … result.0`** **`P.Negative := A.7`** **`if A = 0 then`** **`  P.Zero := 1`** **`else`** **`  P.Zero := 0;`**

`INX`
`INY`
`DEX`
`DEY`
	`X := X + 1`
	`Y := Y + 1`
	`X := X – 1`
	`Y := Y - 1`

If the subtraction results in a borrow (is negative), then the borrow bit is set. However, there is no explicit borrow flag:
If **Carry flag = 1**, then **Borrow = 0**.
If **Carry flag = 0**, then **Borrow = 1**.
(both for addition and subtraction, if the **carry flag is set**, the **output is one more** than if the carry flag is clear.)

(Rephrased from http://www.righto.com/2012/12/the-6502-overflow-flag-explained.html)

Department of Distributed and Dependable Systems D3S

Integer Operations – Subtraction?
Subtract with Carry (= Not Borrow)

SBC X	A – X
A – X – B	Subtract X and Borrow from A

```
A := result.7 … result.0

P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

SBC imm/addr	result := A – imm/addr – NOT(P.Carry)
	P.Carry := NOT(result.7)
	A := result.7 … result.0
	
	P.Negative := A.7
	
	if A = 0 then
	  P.Zero := 1
	else
	  P.Zero := 0;

INX INY DEX DEY	X := X + 1 Y := Y + 1 X := X - 1 Y := Y - 1

If the subtraction results in a borrow (is negative), then the borrow bit is set. However, there is no explicit borrow flag:
If **Carry flag = 1**, then **Borrow = 0**.
If **Carry flag = 0**, then **Borrow = 1**.
(both for addition and subtraction, if the **carry flag is set**, the **output is one more** than if the carry flag is clear.)

(Rephrased from http://www.righto.com/2012/12/the-6502-overflow-flag-explained.html)

Department of Distributed and Dependable Systems

Integer Operations – Subtraction?

Subtract with Carry (= Not Borrow)

SBC X	A – X
A – X – B	Subtract X and Borrow from A
trick: = A – X – B + 256	Adding 256 (1 0000 0000) does not change 8 least significant bits

```
P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

`SBC imm/addr`	`result := A - imm/addr - NOT(P.Carry)`
	`P.Carry := NOT(result.7)`
	`A := result.7 … result.0`
	
	`P.Negative := A.7`
	
	`if A = 0 then`
	`  P.Zero := 1`
	`else`
	`  P.Zero := 0;`

`INX`
`INY`
`DEX`
`DEY`
	`X := X + 1`
	`Y := Y + 1`
	`X := X - 1`
	`Y := Y - 1`

If the subtraction results in a borrow (is negative), then the borrow bit is set. However, there is no explicit borrow flag:
If **Carry flag = 1**, then **Borrow = 0**.
If **Carry flag = 0**, then **Borrow = 1**.
(both for addition and subtraction, if the **carry flag is set**, the **output is one more** than if the carry flag is clear.)

(Rephrased from http://www.righto.com/2012/12/the-6502-overflow-flag-explained.html)

Department of Distributed and Dependable Systems

Integer Operations – Subtraction?
Subtract with Carry (= Not Borrow)

SBC X	A – X
A – X – B	Subtract X and Borrow from A
trick: = A – X – B + 256	Adding 256 (1 0000 0000) does not change 8 least significant bits
= A – X – (1 – C) + 256	Use carry flag
= A – X – 1 + C + 256	
= A – X + C + 255	
= A + (255 – X) + C	

`SBC imm/addr`	`result := A - imm/addr - NOT(P.Carry)`
	`P.Carry := NOT(result.7)`
	`A := result.7 … result.0`
	
	`P.Negative := A.7`
	
	`if A = 0 then`
	`  P.Zero := 1`
	`else`
	`  P.Zero := 0;`

`INX`
`INY`
`DEX`
`DEY`
	`X := X + 1`
	`Y := Y + 1`
	`X := X - 1`
	`Y := Y - 1`

If the subtraction results in a borrow (is negative), then the borrow bit is set. However, there is no explicit borrow flag:
If **Carry flag = 1**, then **Borrow = 0**.
If **Carry flag = 0**, then **Borrow = 1**.
(both for addition and subtraction, if the **carry flag is set**, the **output is one more** than if the carry flag is clear.)

(Rephrased from http://www.righto.com/2012/12/the-6502-overflow-flag-explained.html)

Integer Operations – Subtraction?
Subtract with Carry (= Not Borrow)

SBC X	A – X
A – X – B	Subtract X and Borrow from A
trick: = A – X – B + 256	Adding 256 (1 0000 0000) does not change 8 least significant bits
= A – X – (1 – C) + 256	Use carry flag
= A – X – 1 + C + 256	
= A – X + C + 255	
= A + (255 – X) + C	

```
  1 1 1 1 1 1 1 1
-
  1 0 0 1 0 1 1 1
=
  0 1 1 0 1 0 0 0
```

`SBC imm/addr`	`result := A - imm/addr - NOT(P.Carry)`
	`P.Carry := NOT(result.7)`
	`A := result.7 … result.0`
	
	`P.Negative := A.7`
	
	`if A = 0 then`
	`  P.Zero := 1`
	`else`
	`  P.Zero := 0;`

`INX`
`INY`
`DEX`
`DEY`
	`X := X + 1`
	`Y := Y + 1`
	`X := X - 1`
	`Y := Y - 1`

If the subtraction results in a borrow (is negative), then the borrow bit is set. However, there is no explicit borrow flag:
If **Carry flag = 1**, then **Borrow = 0**.
If **Carry flag = 0**, then **Borrow = 1**.
(both for addition and subtraction, if the **carry flag is set**, the **output is one more** than if the carry flag is clear.)

(Rephrased from http://www.righto.com/2012/12/the-6502-overflow-flag-explained.html)

Department of Distributed and Dependable Systems
D3S

Integer Operations – Subtraction?

Subtract with Carry (= Not Borrow)

SBC X	A – X
A – X – B	Subtract X and Borrow from A
trick: = A – X – B + 256	Adding 256 (1 0000 0000) does not change 8 least significant bits
= A – X – (1 – C) + 256	Use carry flag
= A – X – 1 + C + 256	
= A – X + C + 255	
= A + (255 – X) + C	

```
1 1 1 1 1 1 1 1
       -
1 0 0 1 0 1 1 1   NOT
       =
0 1 1 0 1 0 0 0
```

`SBC imm/addr`	`result := A - imm/addr - NOT(P.Carry)`
	`P.Carry := NOT(result.7)`
	`A := result.7 … result.0`
	
	`P.Negative := A.7`
	
	`if A = 0 then`
	`  P.Zero := 1`
	`else`
	`  P.Zero := 0;`

`INX`
`INY`
`DEX`
`DEY`
	`X := X + 1`
	`Y := Y + 1`
	`X := X - 1`
	`Y := Y - 1`

If the subtraction results in a borrow (is negative), then the borrow bit is set. However, there is no explicit borrow flag:
If **Carry flag = 1**, then **Borrow = 0**.
If **Carry flag = 0**, then **Borrow = 1**.
(both for addition and subtraction, if the **carry flag is set**, the **output is one more** than if the carry flag is clear.)

(Rephrased from http://www.righto.com/2012/12/the-6502-overflow-flag-explained.html)

Integer Operations – Subtraction?

Subtract with Carry (= Not Borrow)

SBC X	A – X
A – X – B	Subtract X and Borrow from A
trick: = A – X – B + 256	Adding 256 (1 0000 0000) does not change 8 least significant bits
= A – X – (1 – C) + 256	Use carry flag
= A – X – 1 + C + 256	
= A – X + C + 255	
= A + (255 – X) + C	
= A + NOT(X) + C	255 – X equals NOT(X) [one's complements of X)
α = NOT X = A + α + C	
α = NOT X **ADC α**	

```
1 1 1 1 1 1 1 1
      -
1 0 0 1 0 1 1 1   NOT
      =
0 1 1 0 1 0 0 0
```

```
P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

INX INY DEX DEY	X := X + 1 Y := Y + 1 X := X - 1 Y := Y - 1

If the subtraction results in a borrow (is negative), then the borrow bit is set. However, there is no explicit borrow flag:
If **Carry flag = 1**, then **Borrow = 0**.
If **Carry flag = 0**, then **Borrow = 1**.
(both for addition and subtraction, if the **carry flag is set**, the **output is one more** than if the carry flag is clear.)

(Rephrased from http://www.righto.com/2012/12/the-6502-overflow-flag-explained.html)

Shifts in More Detail

BONUS

ASL A

```
oldA := A
A.0 := 0
A.1 := oldA.0
A.2 := oldA.1
…
A.7 := oldA.6
P.Carry := oldA.7
```

LSR A

```
oldA := A
A.7 := 0
A.6 := oldA.7
A.5 := oldA.6
…
A.0 := oldA.1
P.Carry := oldA.0
```

ROR A

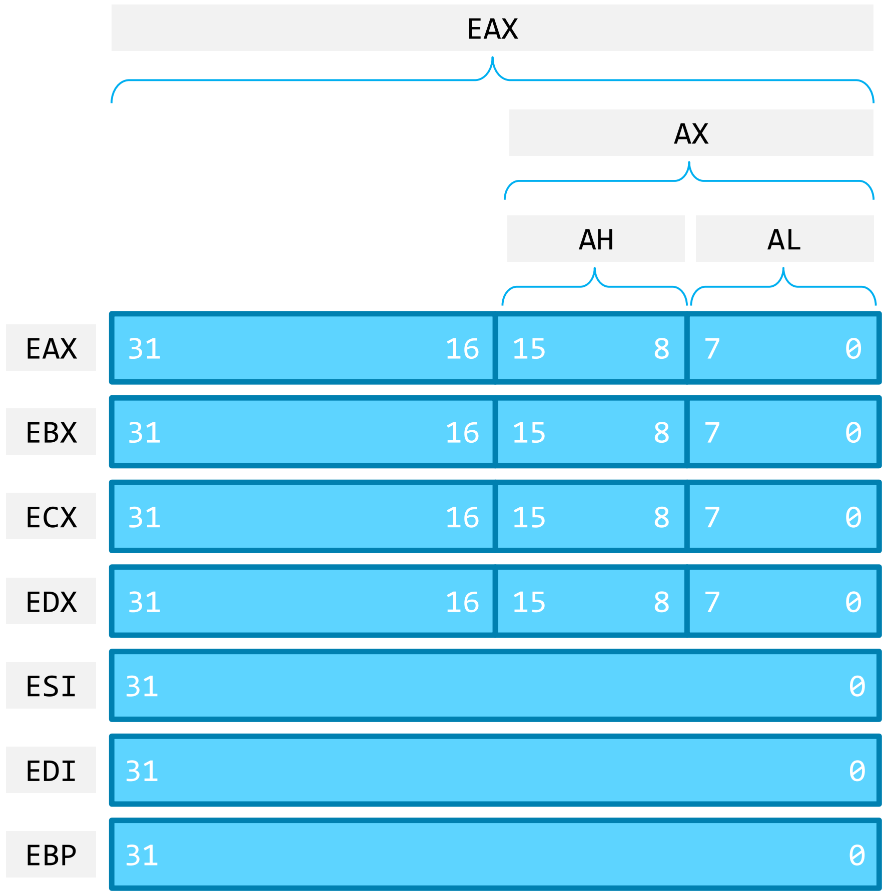
```
oldA := A
A.7 := P.Carry
A.6 := oldA.7
A.5 := oldA.6
…
A.0 := oldA.1
P.Carry := oldA.0
```

ROL A

Using Carry flag as “ninth” bit as well

```
P.Negative := A.7

if A = 0 then
  P.Zero := 1
else
  P.Zero := 0;
```

X86-* Registers: More Registers → More Freedom For Programmer/Compiler



x86 (IA-32)

x86: OP *target, source*

```
MOV target, source → target := source

OP target, source → target := target OP source
```

x86: OP *target, source*

```
LD? #$xx
LD? $xxxx
LD? $xxxx,?

ST? $xxxx
ST? $xxxx,X

T??
```

```
MOV r, imm
MOV r, [addr]
MOV r2, [r1 + addr]

MOV [addr], r
MOV [r1 + addr], r2

MOV r2, r1
```

MOV *target, source* → *target* := *source*

OP *target, source* → *target* := *target* OP *source*

x86: OP *target, source*

```
LD? #$xx
LD? $xxxx
LD? $xxxx,?

ST? $xxxx
ST? $xxxx,X

T??
```

```
MOV r, imm
MOV r, [addr]
MOV r2, [r1 + addr]

MOV [addr], r
MOV [r1 + addr], r2

MOV r2, r1
```

Most complex x86 addressing mode:

```
[r1 + (r2 * scale) + imm]
```

scale = immediate value: 1, 2, 4, 8

MOV *target, source* → *target* := *source*

OP *target, source* → *target* := *target* OP *source*

x86: OP *target, source*

```
LD? #$xx
LD? $xxxx
LD? $xxxx,?

ST? $xxxx
ST? $xxxx,X

T??
```

```
MOV r, imm
MOV r, [addr]
MOV r2, [r1 + addr]

MOV [addr], r
MOV [r1 + addr], r2

MOV r2, r1

ADD r, reg/imm/addr

ADC r, reg/imm/addr

SUB r, reg/imm/addr
SBB r, reg/imm/addr

IMUL r, reg/imm/addr
IDIV r, reg/imm/addr

OR r, reg/imm/addr
AND r, reg/imm/addr
XOR r, reg/imm/addr
NOT r

SHR reg/addr, imm/CL
SAR reg/addr, imm/CL
SHL reg/addr, imm/CL
```

Most complex x86 addressing mode:

`[r1 + (r2 * scale) + imm]`

scale = immediate value: 1, 2, 4, 8

```
r := r + reg/imm/addr
  EFlags.Carry := result.32

r := r + reg/imm/addr + EFlags.Carry
  EFlags.Carry := result.32
```

MOV *target, source* → *target* := *source*

OP *target, source* → *target* := *target* OP *source*

x86: Flags

```
LD? #$xx
LD? $xxxx
LD? $xxxx,?

ST? $xxxx
ST? $xxxx,X

T??
```

```
MOV r, imm
MOV r, [addr]
MOV r2, [r1 + addr]

MOV [addr], r
MOV [r1 + addr], r2

MOV r2, r1

ADD r, reg/imm/addr

ADC r, reg/imm/addr

SUB r, reg/imm/addr
SBB r, reg/imm/addr

IMUL r, reg/imm/addr
IDIV r, reg/imm/addr

OR r, reg/imm/addr
AND r, reg/imm/addr
XOR r, reg/imm/addr
NOT r

SHR reg/addr, imm/CL
SAR reg/addr, imm/CL
SHL reg/addr, imm/CL
```

Most complex x86 addressing mode:

```
[r1 + (r2 * scale) + imm]
```

scale = immediate value: 1, 2, 4, 8

```
EFlags.Sign := target.31

if target = 0 then
  EFlags.Zero := 1
else
  EFlags.Zero := 0;
```

```
r := r + reg/imm/addr
  EFlags.Carry := result.32

r := r + reg/imm/addr + EFlags.Carry
  EFlags.Carry := result.32
```

```
CLC

STC
```

```
EFlags.Carry := 0

EFlags.Carry := 1
```

Department of Distributed and Dependable Systems

Complex Expressions

6502 CPU: **r = a + b + e – (c + d)**

8-bit variables

6502 CPU: **r = a + b + e – (c + d)**

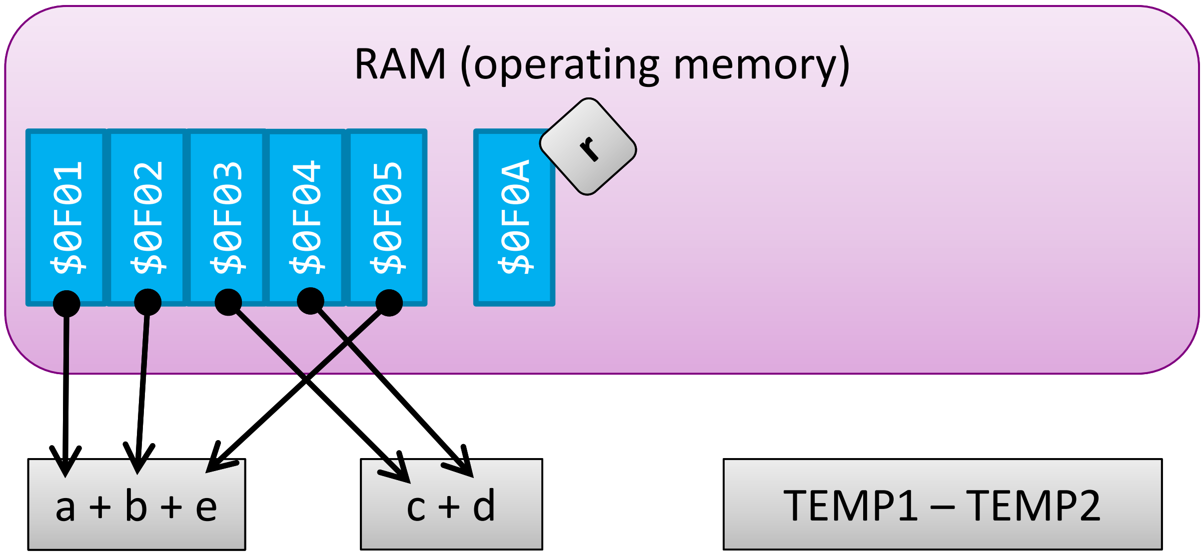
8-bit variables

TEMP1 = a + b + e
TEMP2 = c + d
r = TEMP1 – TEMP2

6502 CPU: **r = a + b + e – (c + d)**

8-bit variables

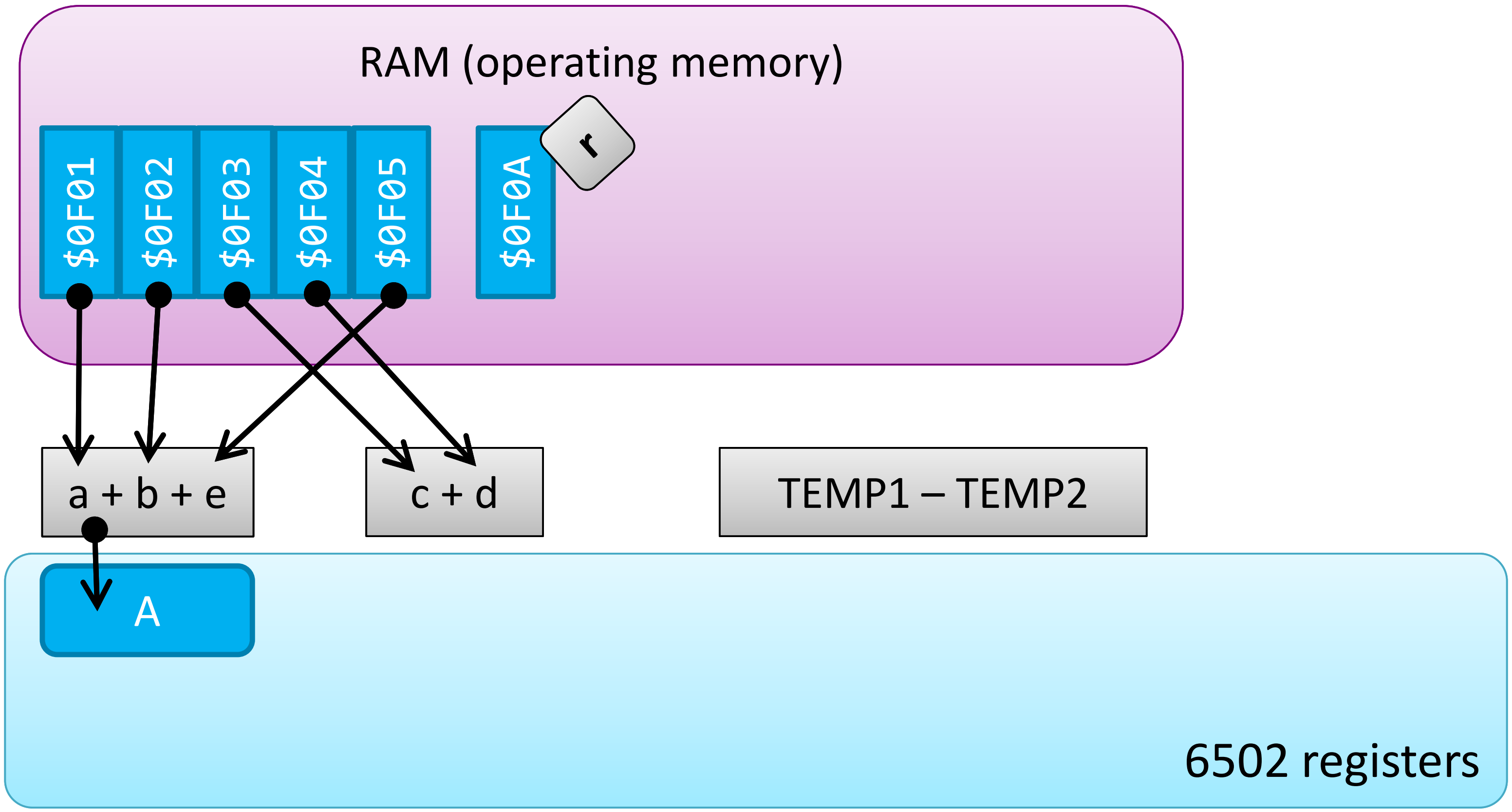
TEMP1 = a + b + e
TEMP2 = c + d
r = TEMP1 – TEMP2



6502 CPU: **r = a + b + e – (c + d)**

8-bit variables

TEMP1 = a + b + e
TEMP2 = c + d
r = TEMP1 – TEMP2

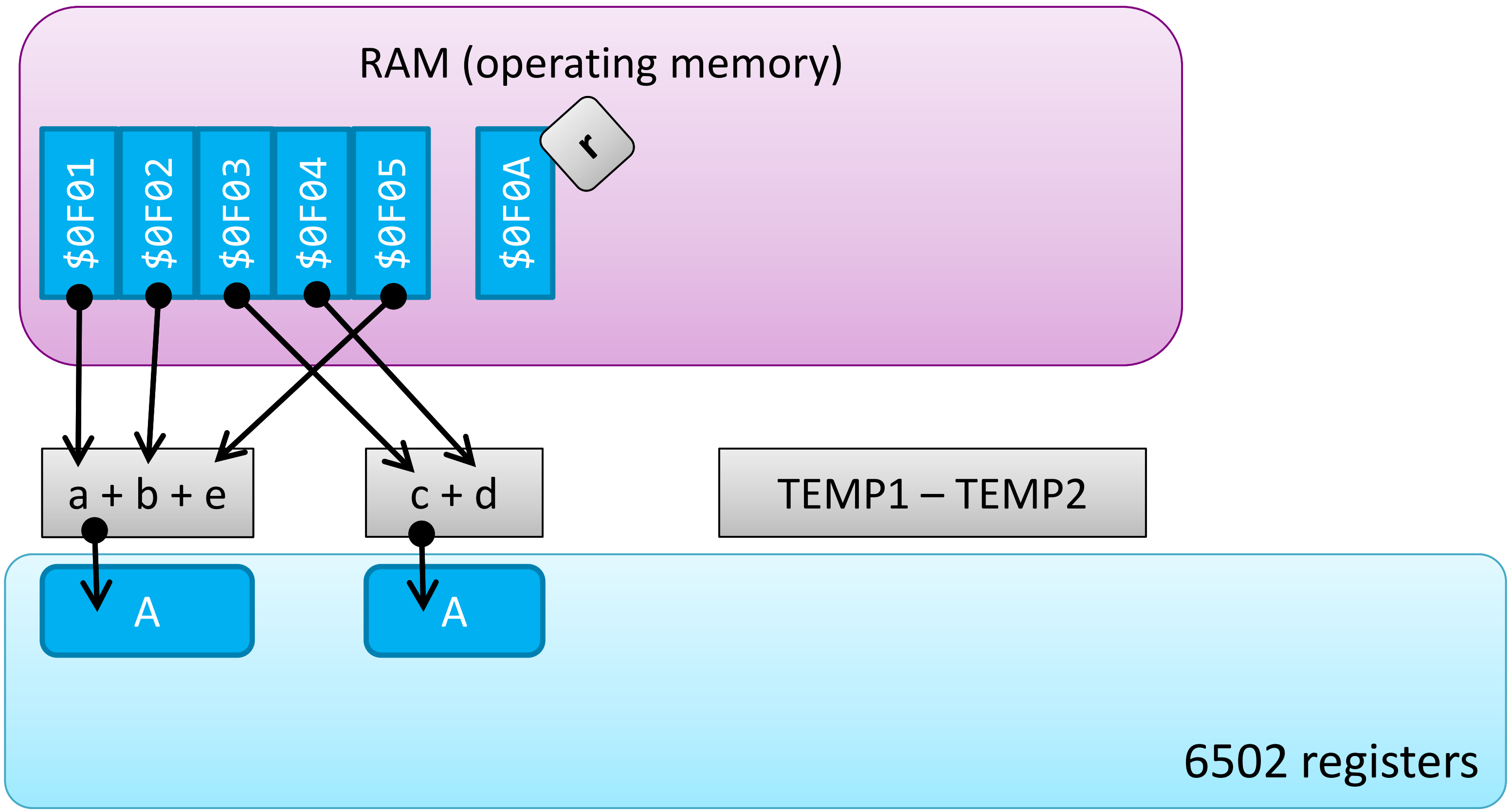


Department of Distributed and Dependable Systems

6502 CPU: r = a + b + e – (c + d)

8-bit variables

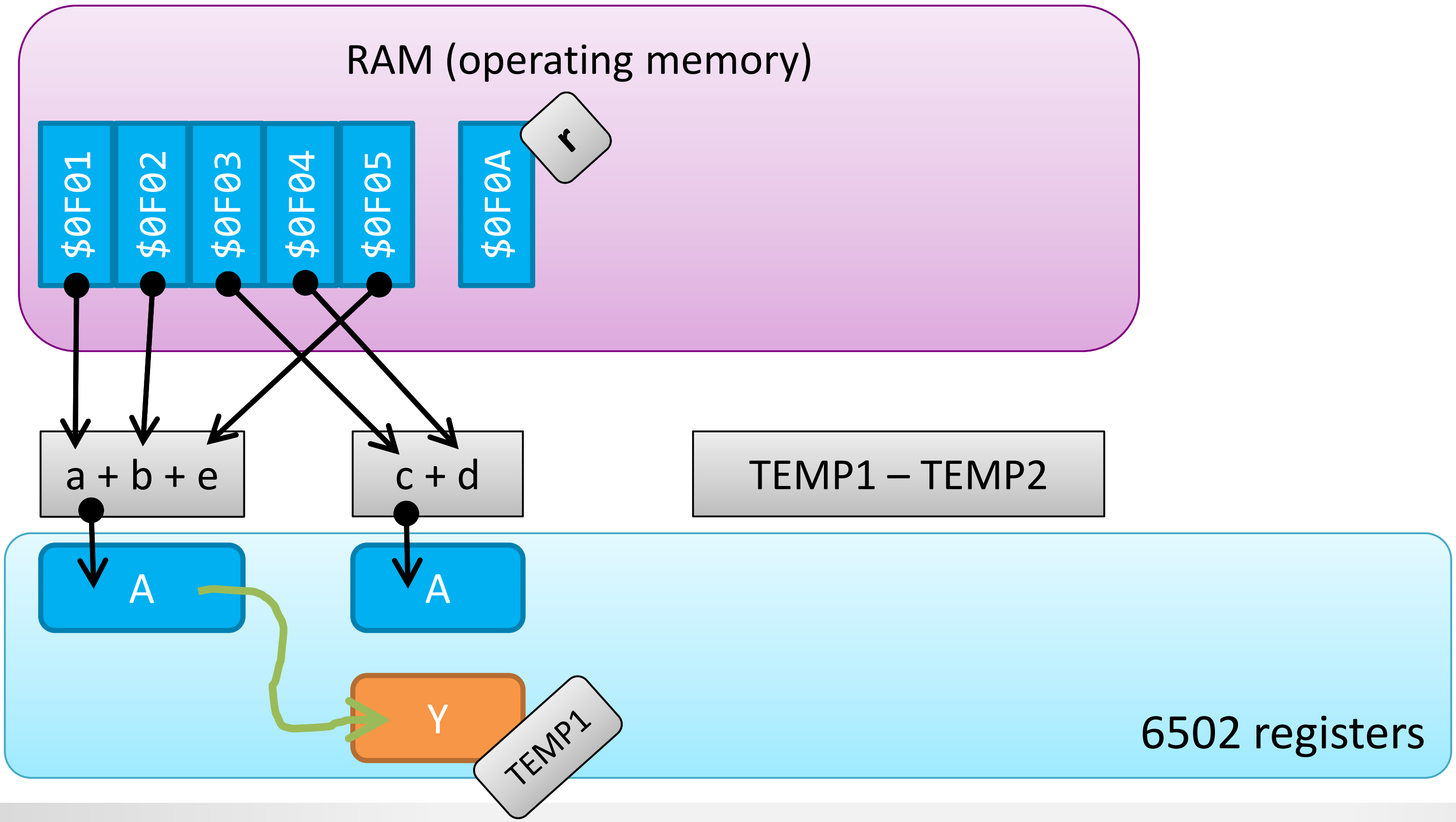
TEMP1 = a + b + e
TEMP2 = c + d
r = TEMP1 – TEMP2



6502 CPU: **r = a + b + e – (c + d)**

8-bit variables

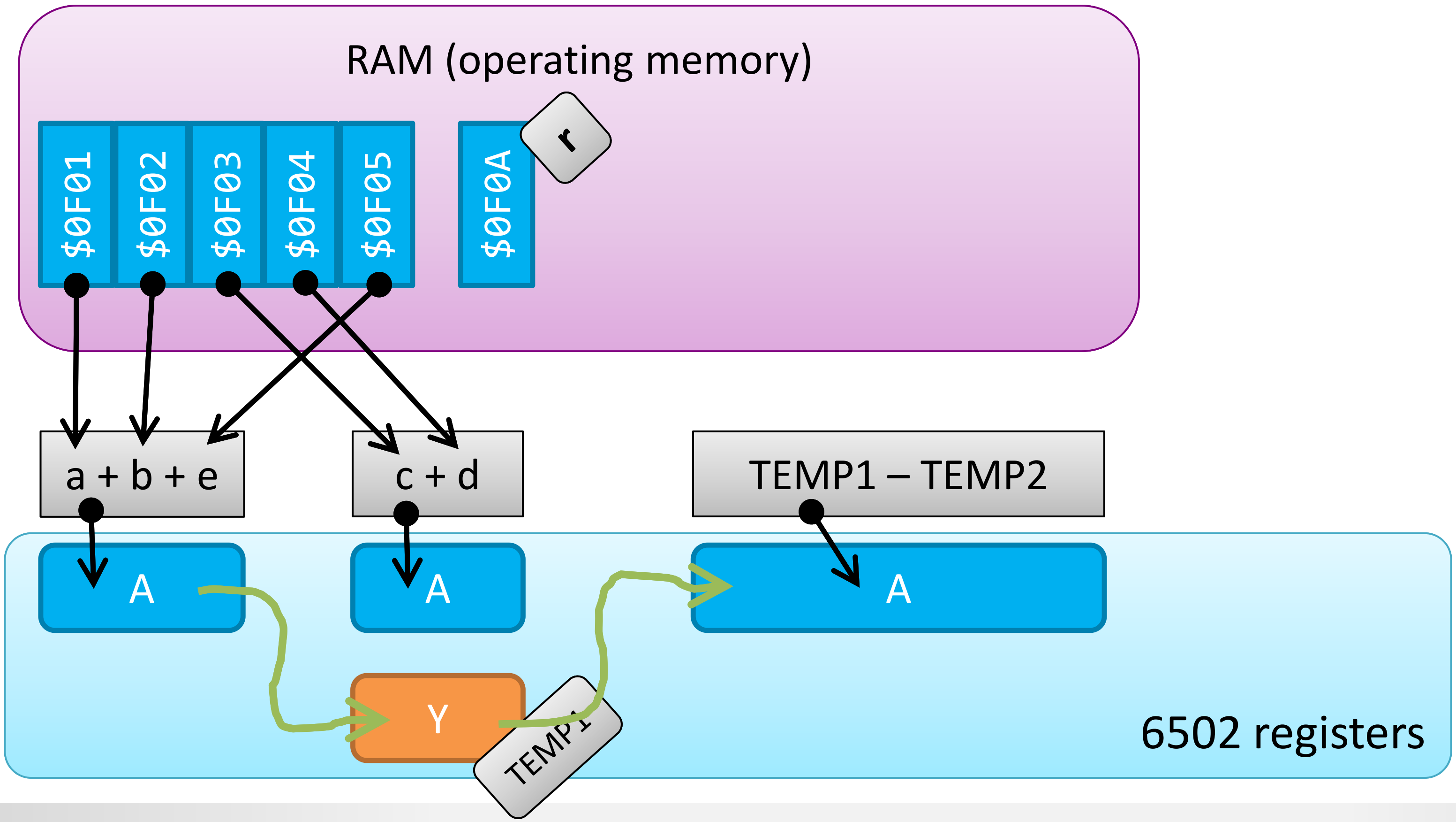
TEMP1 = a + b + e
TEMP2 = c + d
r = TEMP1 – TEMP2



6502 CPU: **r = a + b + e – (c + d)**

8-bit variables

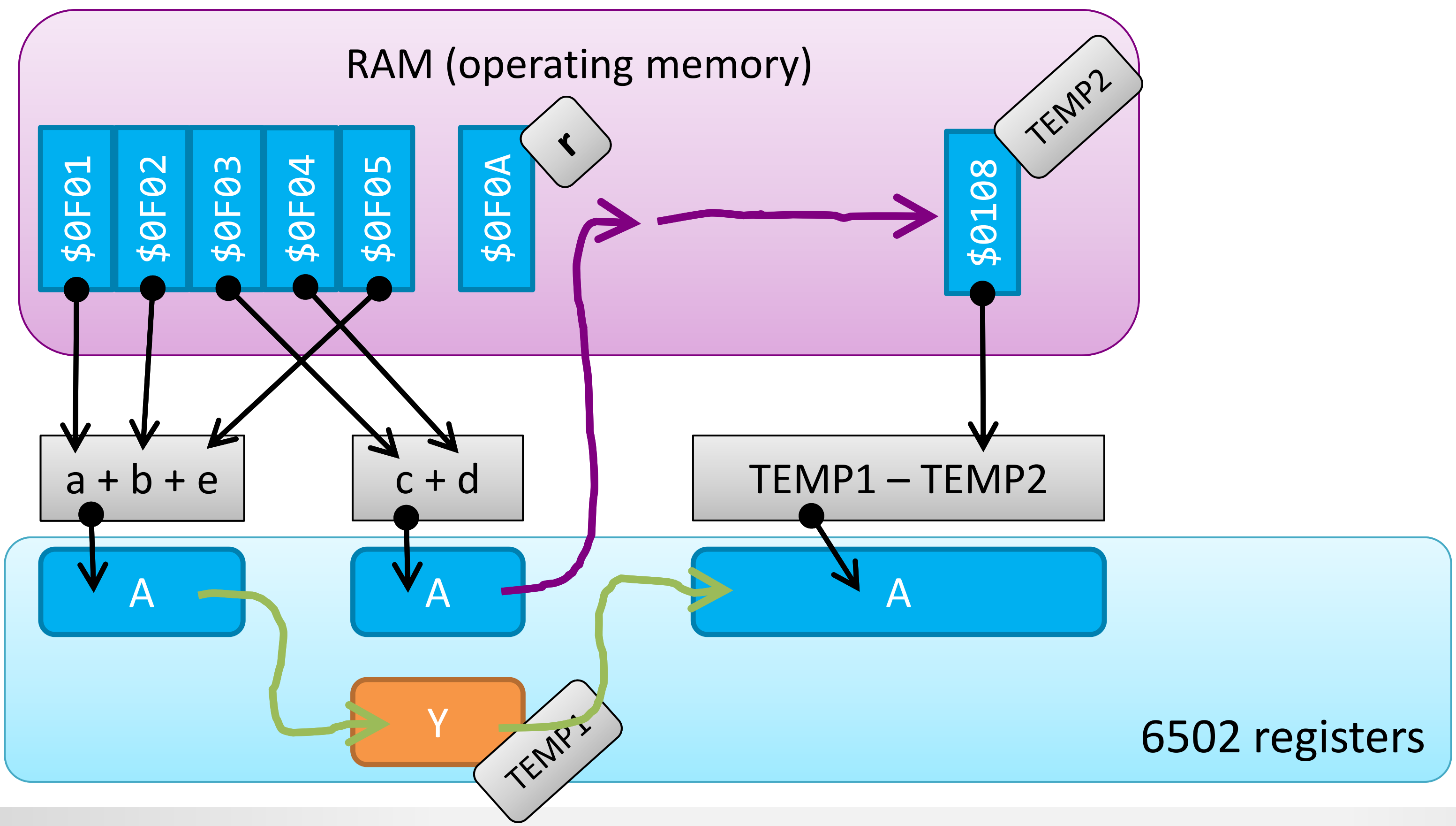
TEMP1 = a + b + e
TEMP2 = c + d
r = TEMP1 – TEMP2



6502 CPU: **r = a + b + e – (c + d)**

8-bit variables

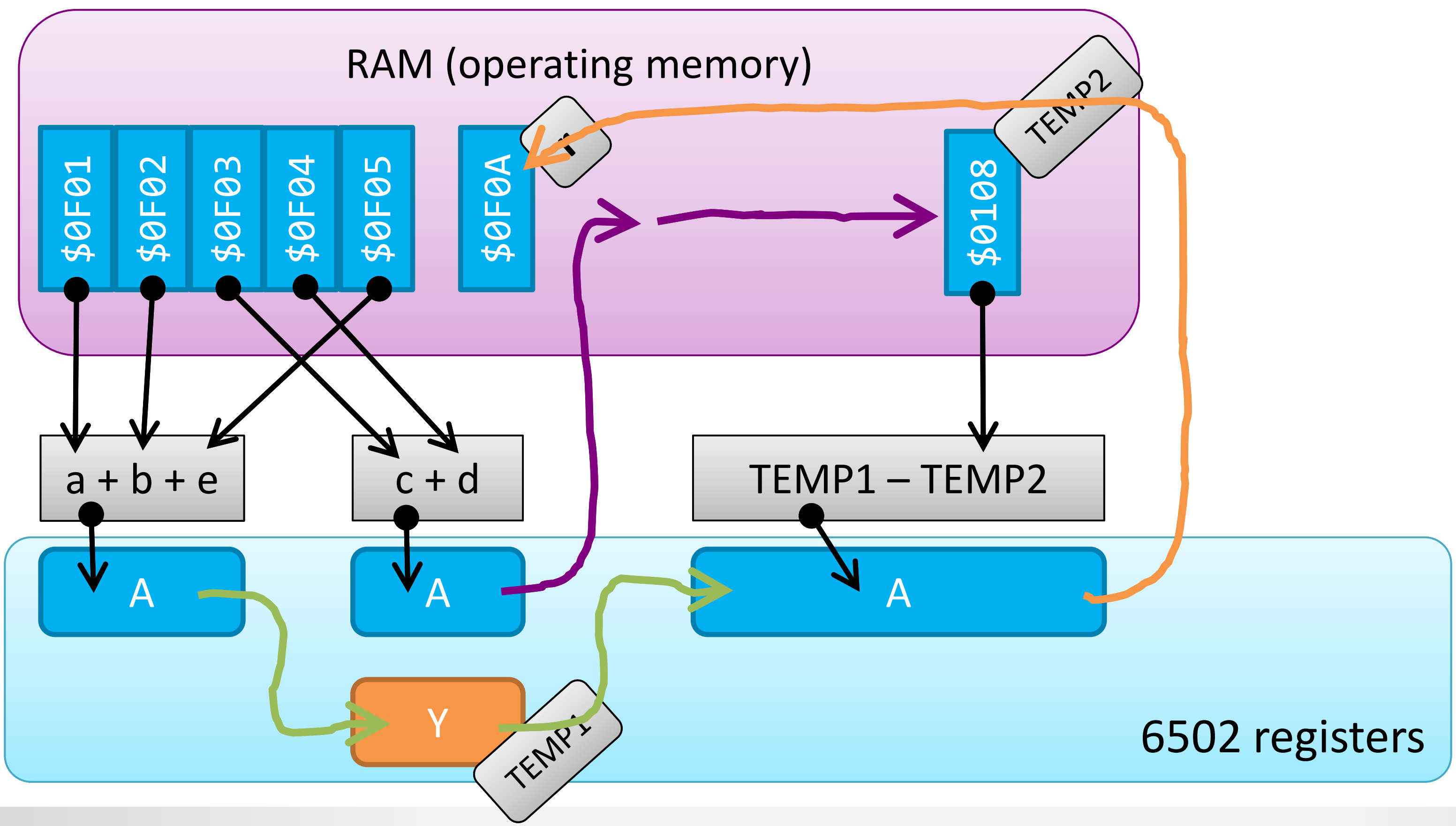
TEMP1 = a + b + e
TEMP2 = c + d
r = TEMP1 – TEMP2



6502 CPU: r = a + b + e – (c + d)

8-bit variables

TEMP1 = a + b + e
TEMP2 = c + d
r = TEMP1 – TEMP2

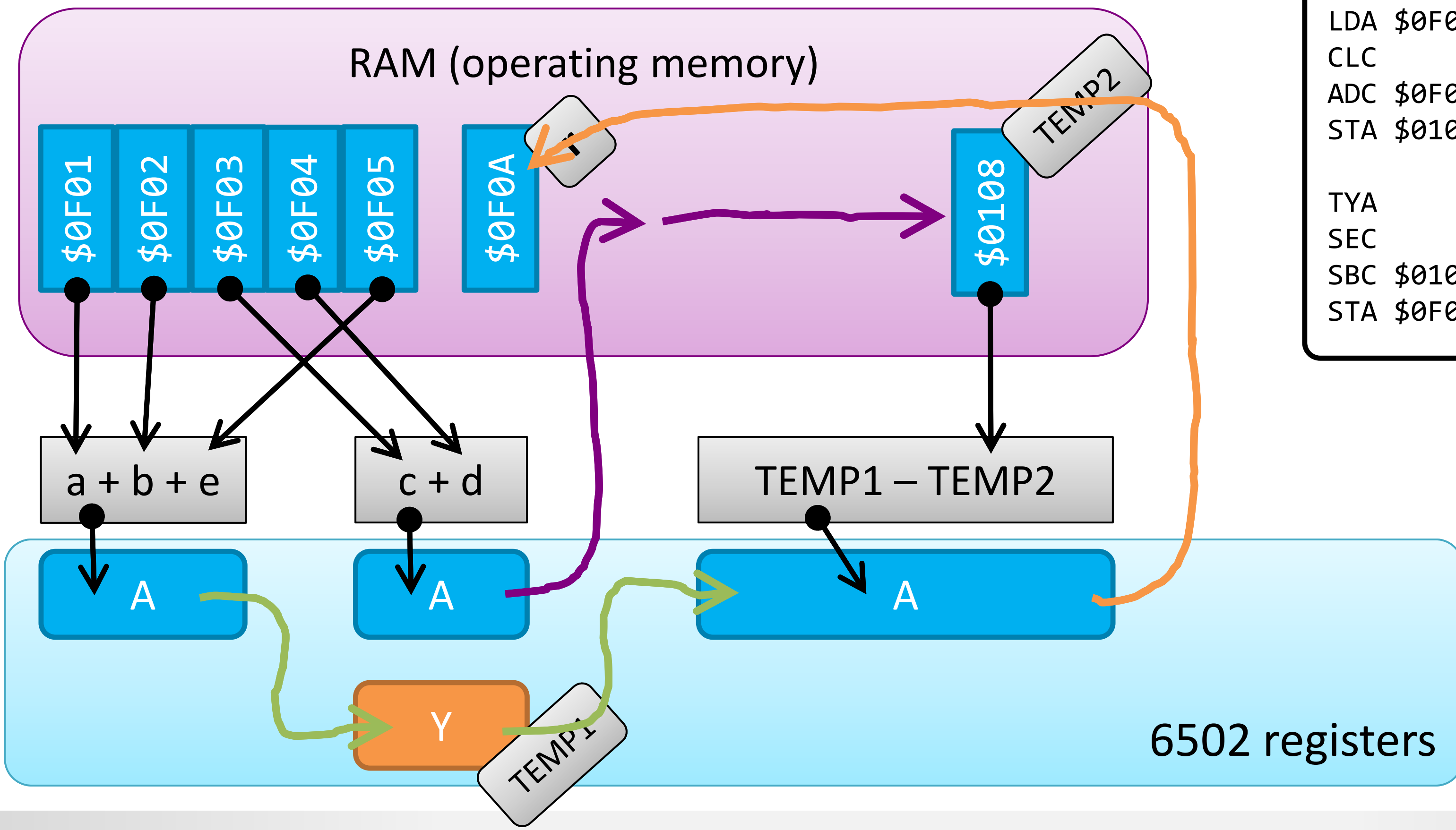


6502 CPU: **r = a + b + e – (c + d)**

8-bit variables

TEMP1 = a + b + e
TEMP2 = c + d
r = TEMP1 – TEMP2

```
LDA $0F01
CLC
ADC $0F02
CLC
ADC $0F05
TAY

LDA $0F03
CLC
ADC $0F04
STA $0108

TYA
SEC
SBC $0108
STA $0F0A
```

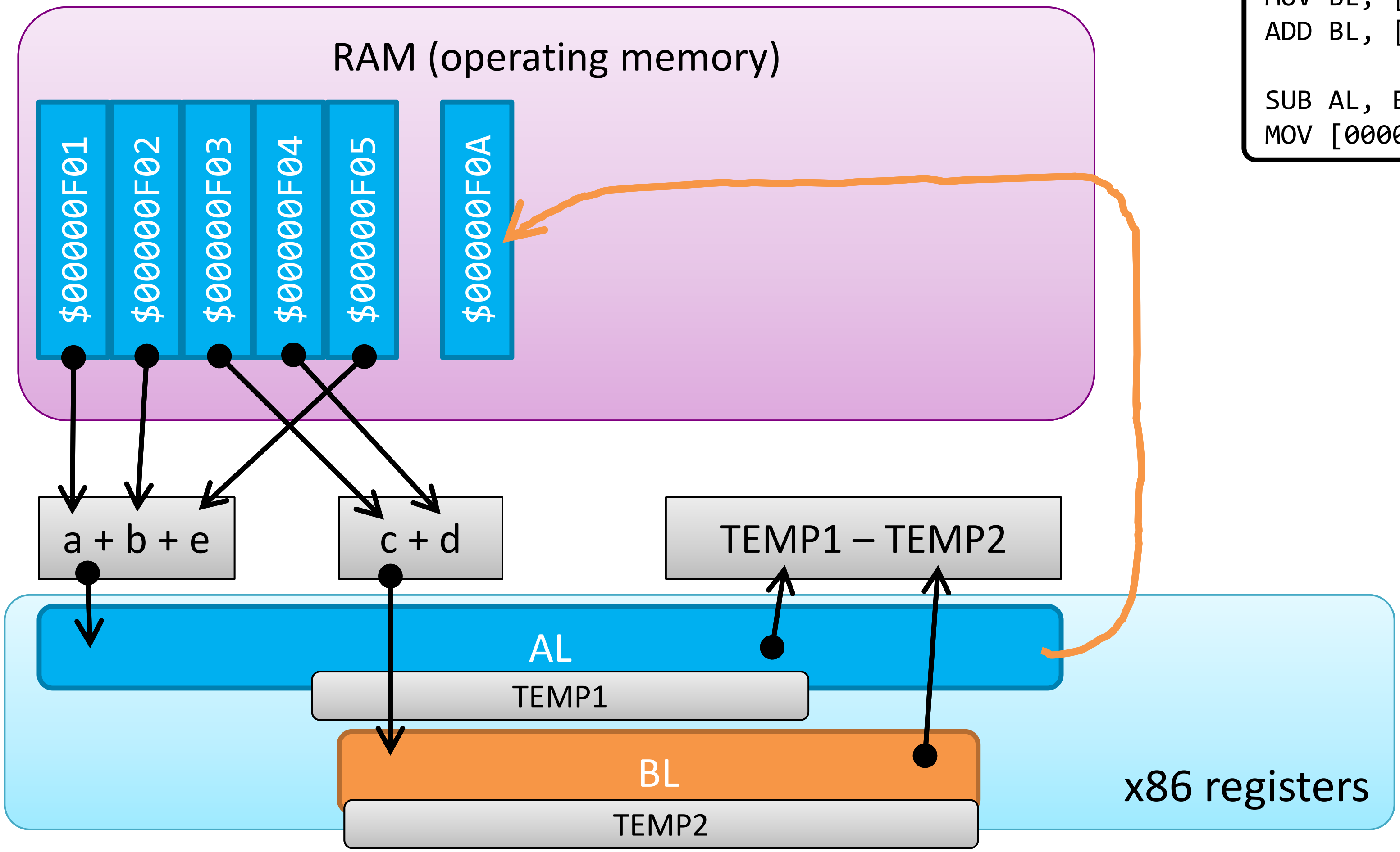


x86 CPU: r = a + b + e – (c + d)

8-bit variables

TEMP1 = a + b + e
TEMP2 = c + d
r = TEMP1 – TEMP2

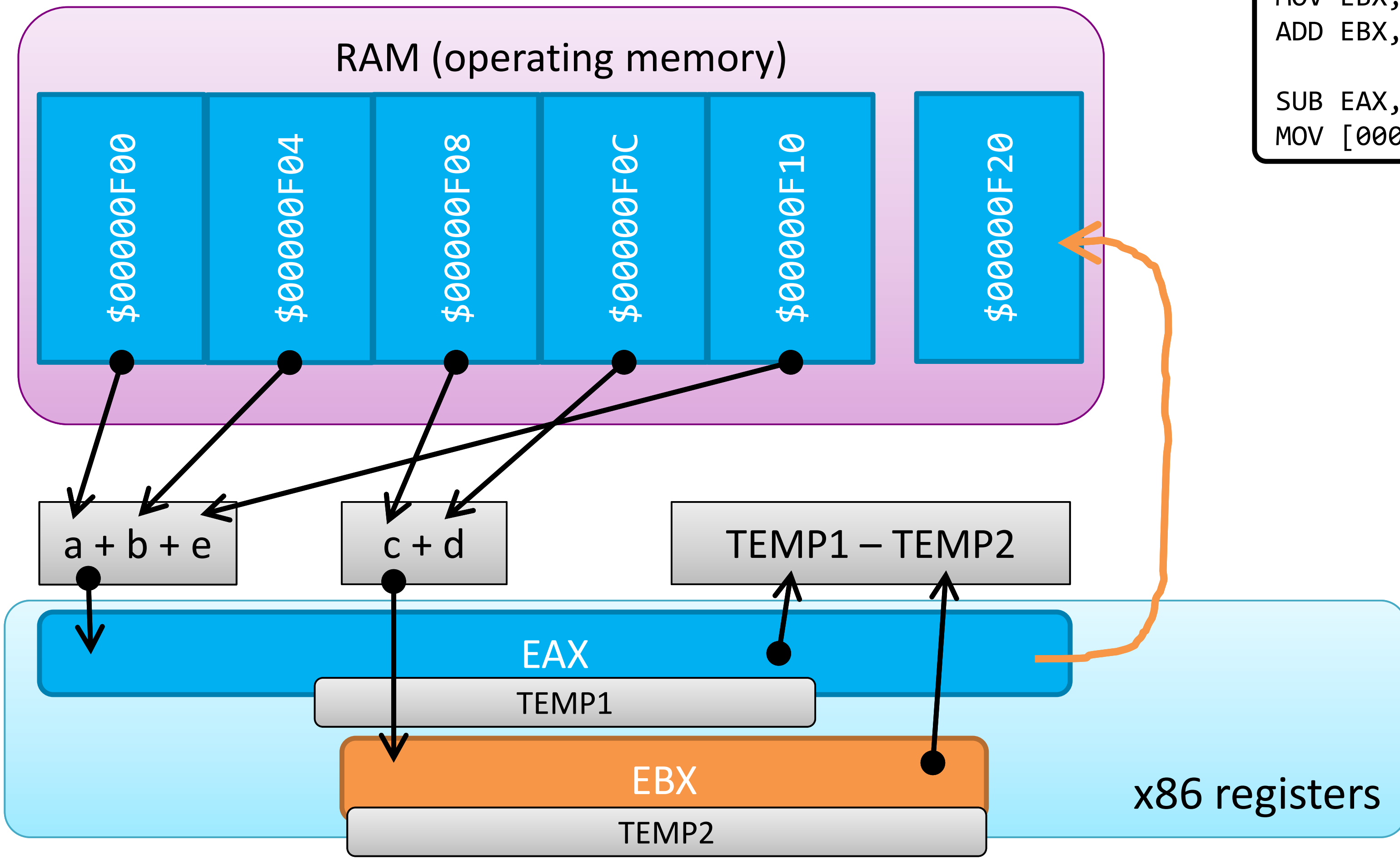
```
MOV AL, [00000F01h]
ADD AL, [00000F02h]
ADD AL, [00000F05h]

MOV BL, [00000F03h]
ADD BL, [00000F04h]

SUB AL, BL
MOV [00000F0Ah], AL
```



x86 CPU: r = a + b + e – (c + d)

32-bit variables

TEMP1 = a + b + e
TEMP2 = c + d
r = TEMP1 – TEMP2




```
MOV EAX, [00000F00h]
ADD EAX, [00000F04h]
ADD EAX, [00000F10h]

MOV EBX, [00000F08h]
ADD EBX, [00000F0Ch]

SUB EAX, EBX
MOV [00000F20h], EAX
```