CS 33

Data Representation (Part 4)

Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Floating-Point Operations: Basic Idea

- $x +_f y = \text{Round}(x + y)$

- $x \times_f y = \text{Round}(x \times y)$

- Basic idea
  - first compute exact result
  - make it fit into desired precision
    - » possibly overflow if exponent too large
    - » possibly round to fit into `frac`

Rounding

- Rounding modes (illustrated with $ rounding)

	$1.40	$1.60	$1.50	$2.50	–$1.50
towards zero	$1	$1	$1	$2	–$1
round down (−∞)	$1	$1	$1	$2	–$2
round up (+∞)	$2	$2	$2	$3	–$1
nearest integer	$1	$2	?	?	?
nearest even (default)	$1	$2	$2	$2	–$2

Creating a Floating Point Number

s	exp	frac
1	4-bits	3-bits

- **Steps**
  - normalize to have leading 1
  - round to fit within fraction
  - postnormalize to deal with effects of rounding

- **Case study**
  - convert 8-bit unsigned numbers to tiny floating-point format

example numbers

128	10000000
13	00001101
33	00010001
35	00010011
138	10001010
63	00111111

Normalize

s	exp	frac
1	4-bits	3-bits

- **Requirement**
  - – set binary point so that numbers of form 1.xxxxx
  - – adjust all to have leading one
    - » decrement exponent as shift left

Value	Binary	Fraction	Exponent
128	10000000	1.0000000	7
13	00001101	1.1010000	3
17	00010001	1.0001000	4
19	00010011	1.0011000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5

Rounding

1.BBGRXXX

Guard bit: LSB of result

Round bit: 1st bit removed

Sticky bit: OR of remaining bits

- **Round-up conditions**
  - round = 1, sticky = 1 ⇒ > 0.5
  - guard = 1, round = 1, sticky = 0 ⇒ round up to even

Value	Fraction	GRS	Incr?	Rounded
128	1.0000000	000	N	1.000
13	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	111	Y	10.000

Postnormalize

- **Issue**
  - rounding may have caused overflow
  - handle by shifting right once & incrementing exponent

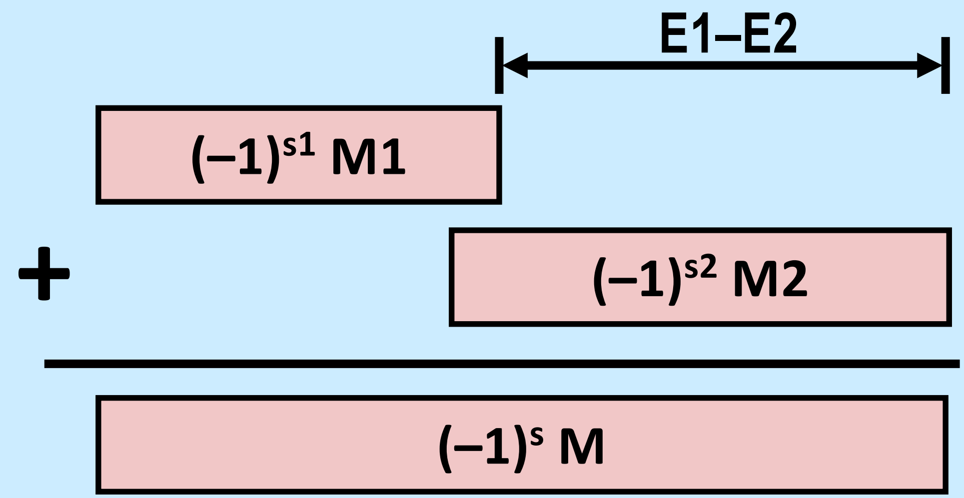
Value	Rounded	Exp	Adjusted	Result
128	1.000	7		128
13	1.101	3		13
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	$1.000*2^6$	64

Floating-Point Multiplication

- $(-1)^{s1}\ M1\ \ 2^{E1}\quad x\quad (-1)^{s2}\ M2\ \ 2^{E2}$
- Exact result: $(-1)^{s}\ M\ \ 2^{E}$
  - sign s: s1 ^ s2
  - significand M: M1 x M2
  - exponent E: E1 + E2
- Fixing
  - if M ≥ 2, shift M right, increment E
  - if E out of range, overflow (or underflow)
  - round M to fit `frac` precision
- Implementation
  - biggest chore is multiplying significands

Floating-Point Addition

- $(-1)^{s1}\ M1\ \ 2^{E1} \quad + \quad (-1)^{s2}\ M2\ \ 2^{E2}$
  - –**assume E1 > E2**



- **Exact result:** $(-1)^{s}\ M\ \ 2^{E}$
  - –sign s, significand M:
    - » result of signed align & add
  - –exponent E: E1

- **Fixing**
  - –if M ≥ 2, shift M right, increment E
  - –if M < 1, shift M left k positions, decrement E by k
  - –overflow if E out of range
  - –round M to fit **`frac`** precision

Floating Point

- **Single precision (float)**

s	exp	frac
1	8-bits	23-bits

  – **range: $\pm 1.8 \times 10^{-38}$ – $\pm 3.4 \times 10^{38}$, ~7 decimal digits**

- **Double Precision (double)**

s	exp	frac
1	11-bits	52-bits

  – **range: $\pm 2.23 \times 10^{-308}$ – $\pm 1.8 \times 10^{308}$, ~16 decimal digits**

Floating Point in C

- **Conversions/casting**
  - –casting between `int`, `float`, and `double` changes bit representation
  - – `double`/`float` → `int`
    - » truncates fractional part
    - » like rounding toward zero
    - » not defined when out of range or NaN: generally sets to TMin
  - – `int` → `double`
    - » exact conversion, as long as `int` has ≤ 53-bit word size
  - – `int` → `float`
    - » will round according to rounding mode

Quiz 1

Suppose *f*, declared to be a `float`, is assigned the largest possible floating-point positive value (other than +∞). What is the value of *g = f+1.0*?

a) f

b) +∞

c) NaN

d) 0

Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Float is not Rational …

- **Floating addition**
  - commutative: $a +^f b = b +^f a$
    - » yes!
  - associative: $a +^f (b +^f c) = (a +^f b) +^f c$
    - » no!
      - $2 +^f (1e38 +^f -1e38) = 2$
      - $(2 +^f 1e38) +^f -1e38 = 0$

Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Float is not Rational …

- **Multiplication**
  - commutative: $a *^f b = b *^f a$
    - » yes!
  - associative: $a *^f (b *^f c) = (a *^f b) *^f c$
    - » no!
      - $1e37 *^f (1e37 *^f 1e\text{-}37) = 1e37$
      - $(1e37 *^f 1e37) *^f 1e\text{-}37 = +\infty$

Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Float is not Rational …

- More …
  - multiplication distributes over addition:
    $a *^f (b +^f c) = (a *^f b) +^f (a *^f c)$
    - » no!
    - » $1e38 *^f (1e38 +^f -1e38) = 0$
    - » $(1e38 *^f 1e38) +^f (1e38 *^f -1e38) = NaN$
  - loss of significance:

    x=y+1

    z=2/(x-y)

    z==2?
    - » not necessarily!
      - consider y = 1e38

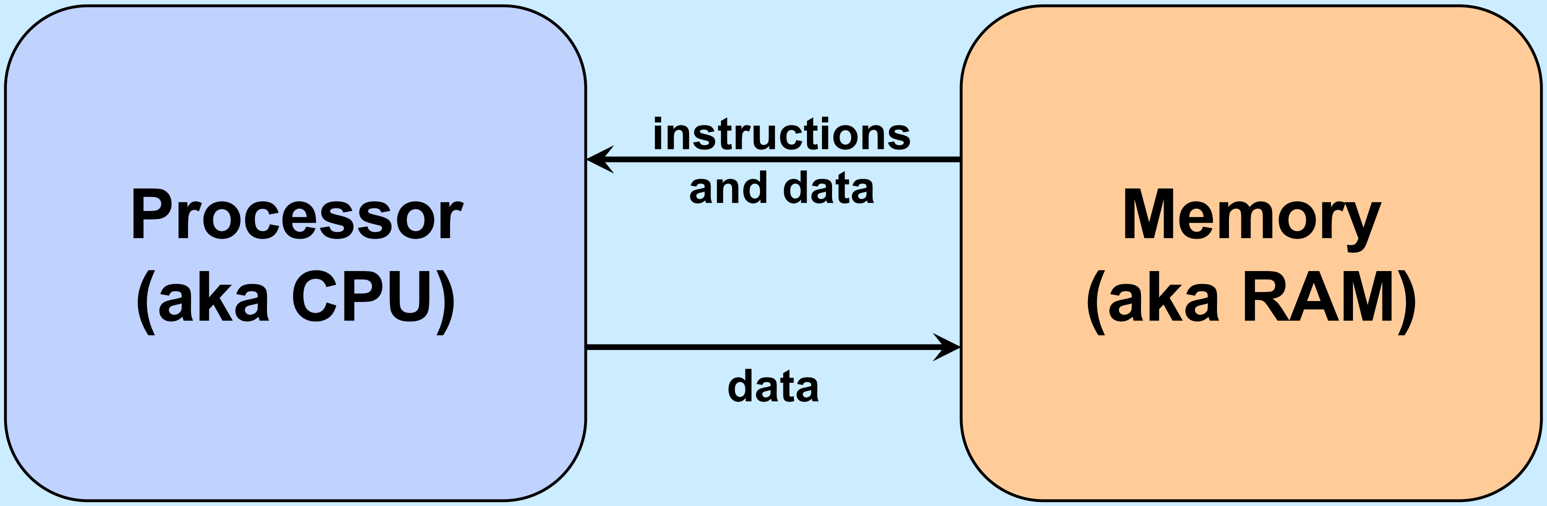
Copyright © 2019 Thomas W. Doeppner. All rights reserved.

CS 33

Intro to Machine Programming

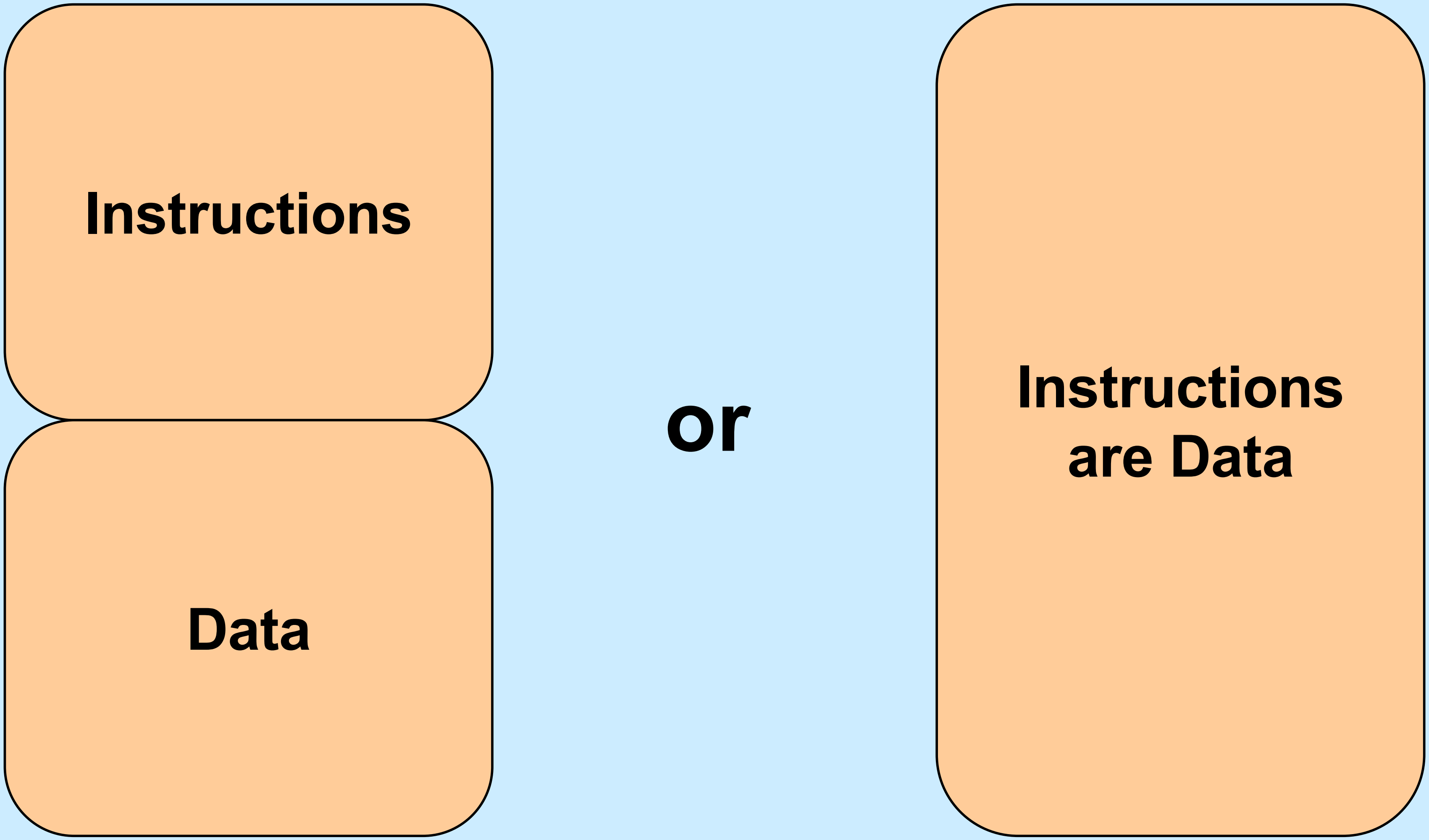
Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Machine Model



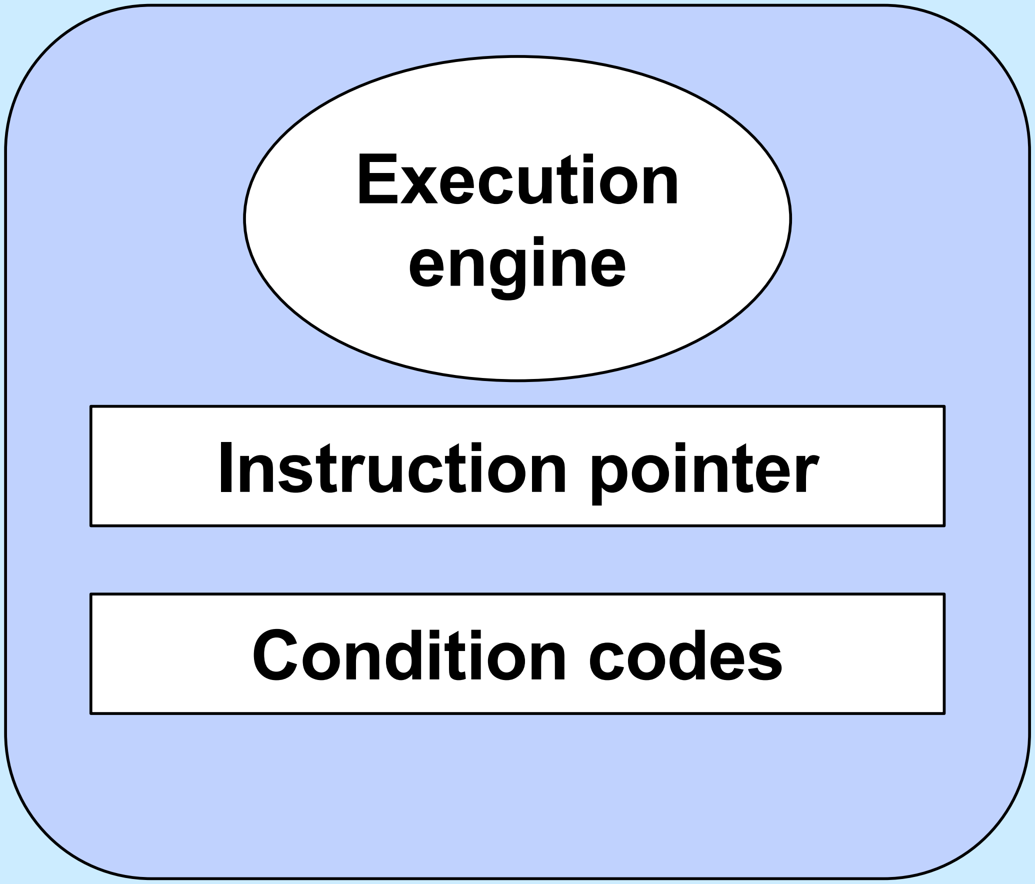
 Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Memory



 Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Processor: Some Details



Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Processor: Basic Operation

```
while (forever) {
  fetch instruction IP points at
  decode instruction
  fetch operands
  execute
  store results
  update IP and condition code
}
```

 Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Instructions ...

Op code	**Operand1**	**Operand2**	**...**

Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Operands

- Form
  – immediate vs. reference
    » value vs. address
- How many?
  – 3
    » add a,b,c
      • c = a + b
  – 2
    » add a,b
      • b += a

 Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Operands (continued)

- Accumulator
  - special memory in the processor
    - known as a *register*
    - fast access
  - allows single-operand instructions
    - add a
      - acc += a
    - add b
      - acc += b

Copyright © 2019 Thomas W. Doeppner. All rights reserved.

From C to Assembler ...

```
a = (b + c) * d;

    mov    b,%acc
    add    c,%acc
    mul    d,%acc
    mov    %acc,a
```

```
if (a<b)
   c = 1;
else
   d = 1;

   cmp   a,b
   jge   .L1
   mov   $1,c
   jmp   .L2
.L1
   mov   $1,d
.L2
```

immediate operand

immediate operand

Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Condition Codes

- **Set of flags giving status of most recent operation:**
  - zero flag
    - » result was zero
  - sign flag
    - » for signed arithmetic interpretation: sign bit is set
  - overflow flag
    - » for signed arithmetic interpretation
  - carry flag (generated by carry or borrow out of most-significant bit)
    - » for unsigned arithmetic interpretation
- **Set implicitly by arithmetic instructions**
- **Set explicitly by compare instruction**
  - cmp a,b
    - » sets flags based on result of b-a

Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Examples (1)

- Assume 32-bit arithmetic

- x is 0x80000000
  - TMIN if interpreted as two's-complement
  - 2^{31} if interpreted as unsigned
- x-1 (0x7fffffff)
  - TMAX if interpreted as two's-complement
  - 2^{31}-1 if interpreted as unsigned
  - zero flag is not set
  - sign flag is not set
  - overflow flag is set
  - carry flag is not set

 Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Examples (2)

- x is 0xffffffff
  - -1 if interpreted as two's-complement
  - UMAX (2^{32}-1) if interpreted as unsigned
- x+1 (0x00000000)
  - zero under either interpretation
  - zero flag is set
  - sign flag is not set
  - overflow flag is not set
  - carry flag is set

Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Examples (3)

- x is 0xffffffff
  - -1 if interpreted as two's-complement
  - UMAX (2^{32}-1) if interpreted as unsigned
- x+2 (0x00000001)
  - (+)1 under either interpretation
  - zero flag is not set
  - sign flag is not set
  - overflow flag is not set
  - carry flag is set

Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Quiz 2

- **Set of flags giving status of most recent operation:**
  - zero flag
    - » result was zero
  - sign flag
    - » for signed arithmetic interpretation: sign bit is set
  - overflow flag
    - » for signed arithmetic interpretation
  - carry flag (generated by carry or borrow out of most-significant bit)
    - » for unsigned arithmetic interpretation
- **Set explicitly by compare instruction**
  - cmp a,b
    - » sets flags based on result of b-a

Which flags are set to one by "cmp 2,1"?

a) overflow flag only
b) carry flag only
c) sign and carry flags only
d) sign and overflow flags only
e) sign, overflow, and carry flags

Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Jump Instructions

- Unconditional jump
  - just do it
- Conditional jump
  - to jump or not to jump determined by condition-code flags
  - field in the op code indicates how this is computed
  - in assembler language, simply say
    - » je
      - jump on equal
    - » jne
      - jump on not equal
    - » jgt
      - jump on greater than
    - » etc.

Copyright © 2019 Thomas W. Doeppner. All rights reserved.

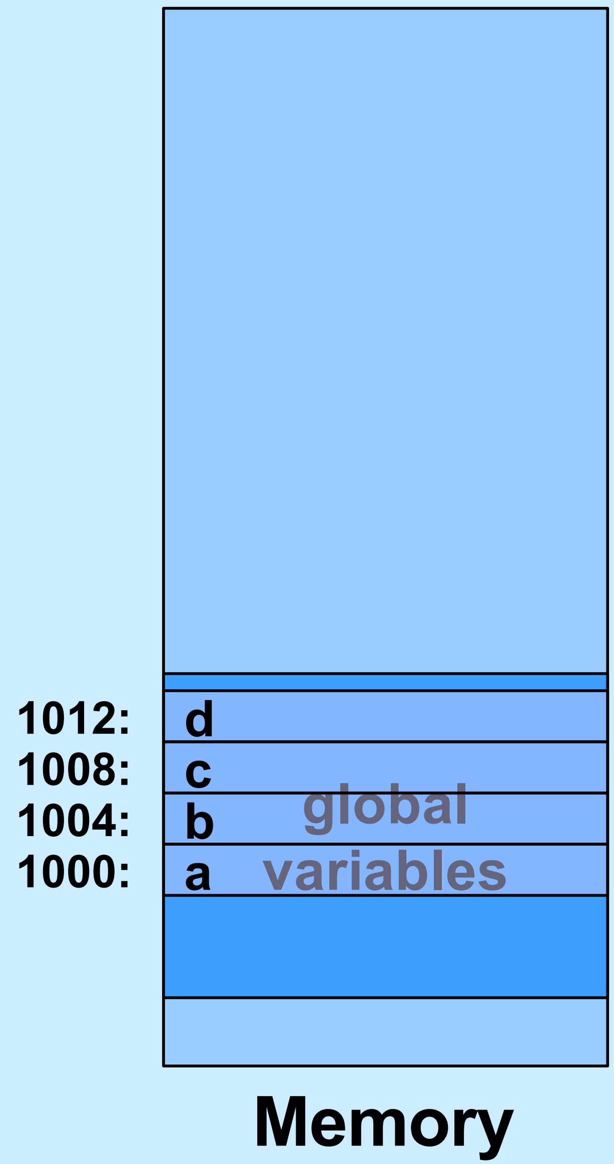
Addresses

```
int a, b, c, d;

int main() {
    a = (b + c) * d;
    ...
}
```

```
mov     b,%acc
add     c,%acc
mul     d,%acc
mov     %acc,a
```

```
mov     1004,%acc
add     1008,%acc
mul     1012,%acc
mov     %acc,1000
```



Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Addresses

```
int b;

int func(int c, int d) {
    int a;
    a = (b + c) * d;
    ...
}

    mov    ?,%acc
    add    ?,%acc
    mul    ?,%acc
    mov    %acc,?
```

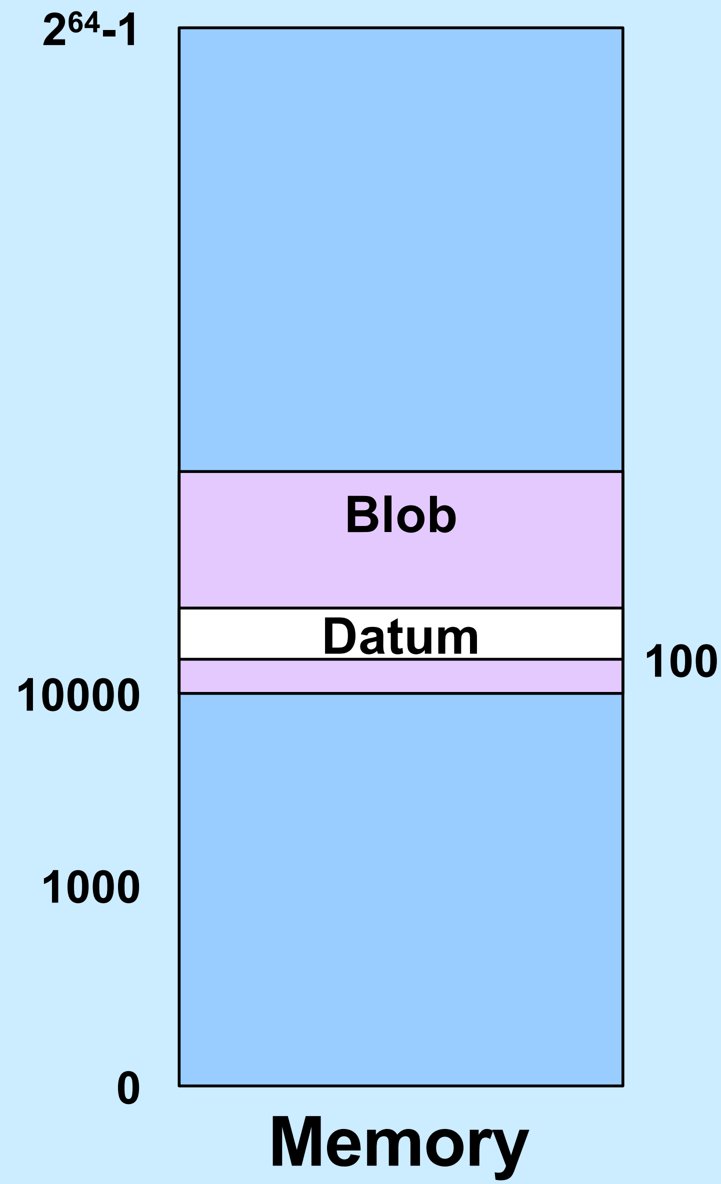
- One copy of *b* for duration of program's execution
  - *b*'s address is the same for each call to *func*
- Different copies of *a*, *c*, and *d* for each call to *func*
  - addresses are different in each call

Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Relative Addresses

- **Absolute address**
  - actual location in memory
- **Relative address**
  - offset from some other location

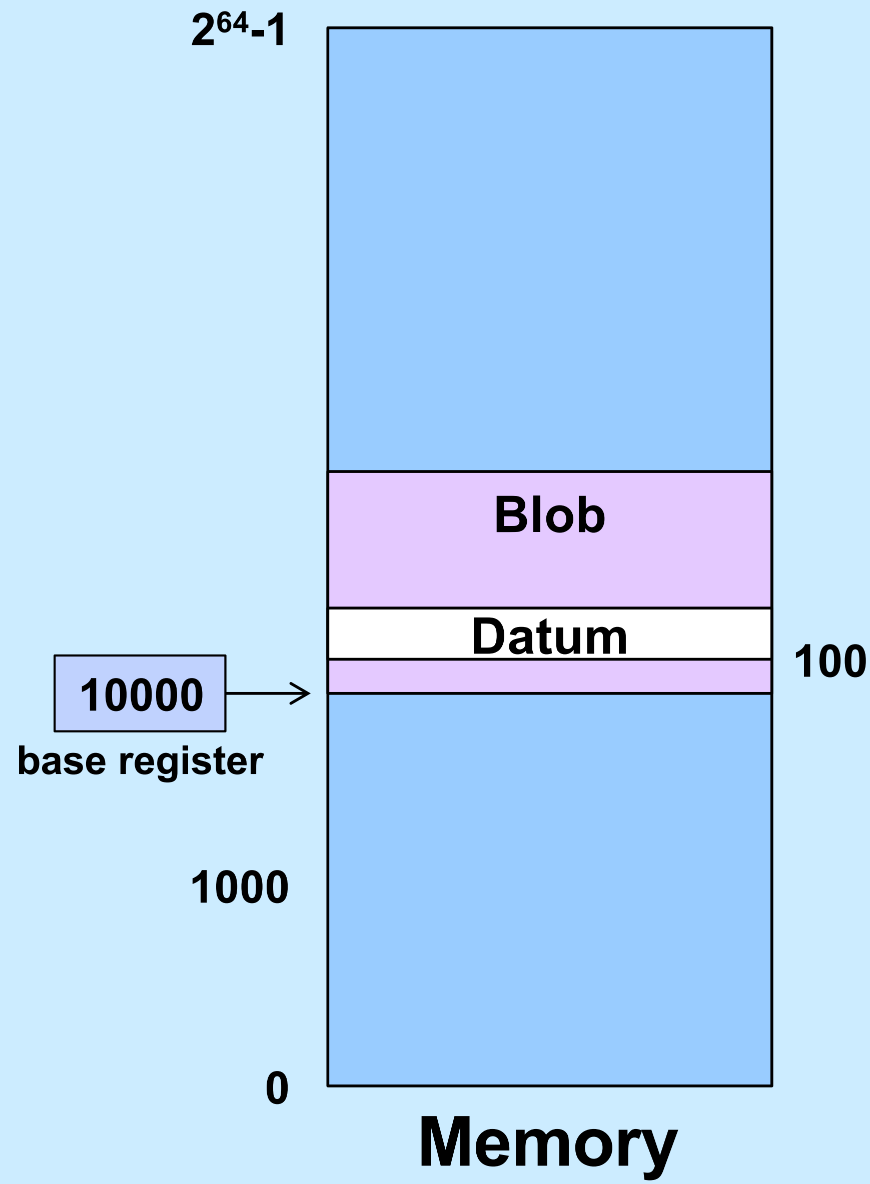
- Blob's absolute address is 10000
- Datum's relative address (to Blob) is 100
  - its absolute address is 10100



Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Base Registers

```
mov $10000, %base
mov $10, 100(%base)
```



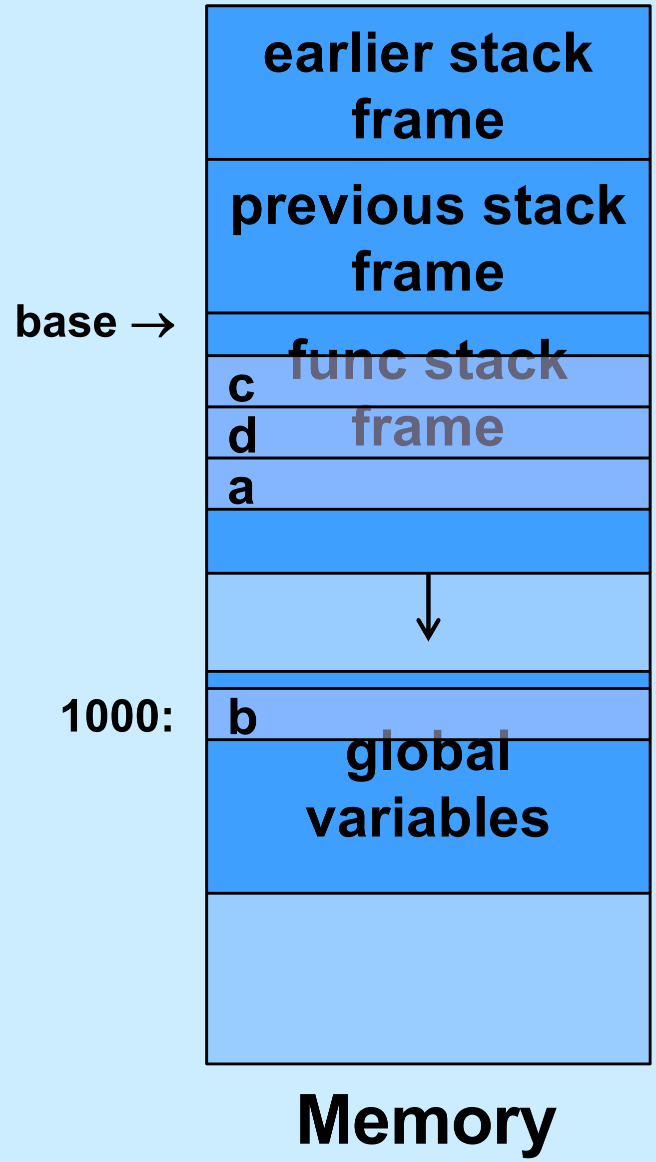
Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Addresses

```
long b;

int func(long c, long d) {
    long a;
    a = (b + c) * d;
    ...
}

    mov    1000,%acc
    add    -8(%base),%acc
    mul    -12(%base),%acc
    mov    %acc,-16(%base)
```



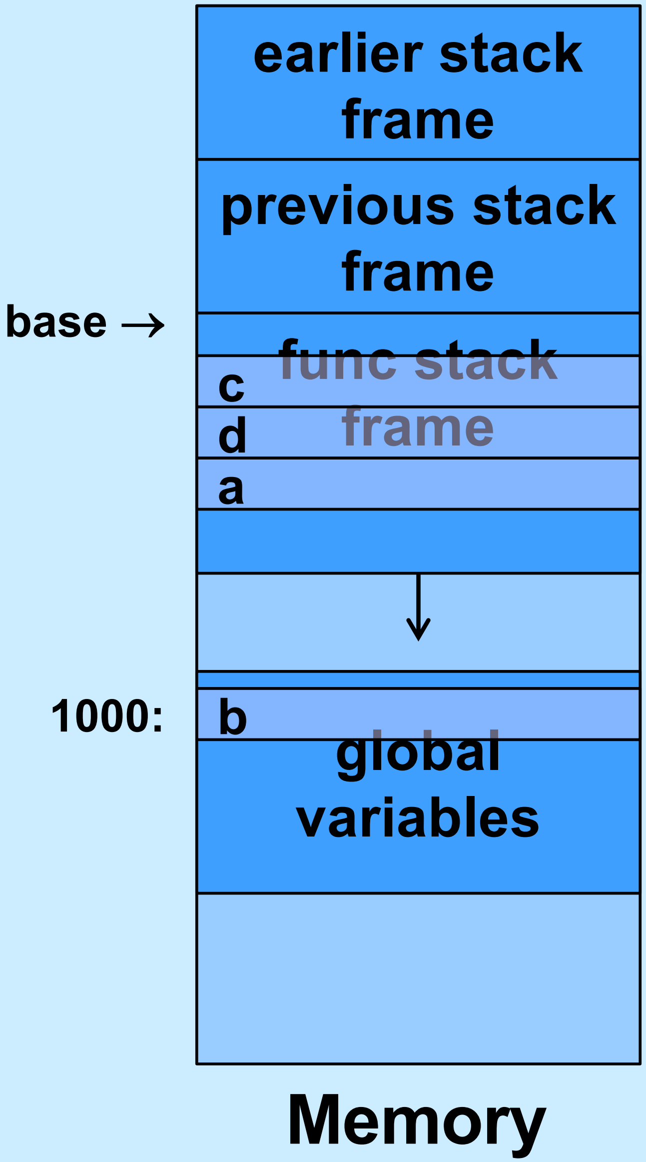
Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Quiz 3

Suppose the value in *base* is 10,000. What is the address of *c*?

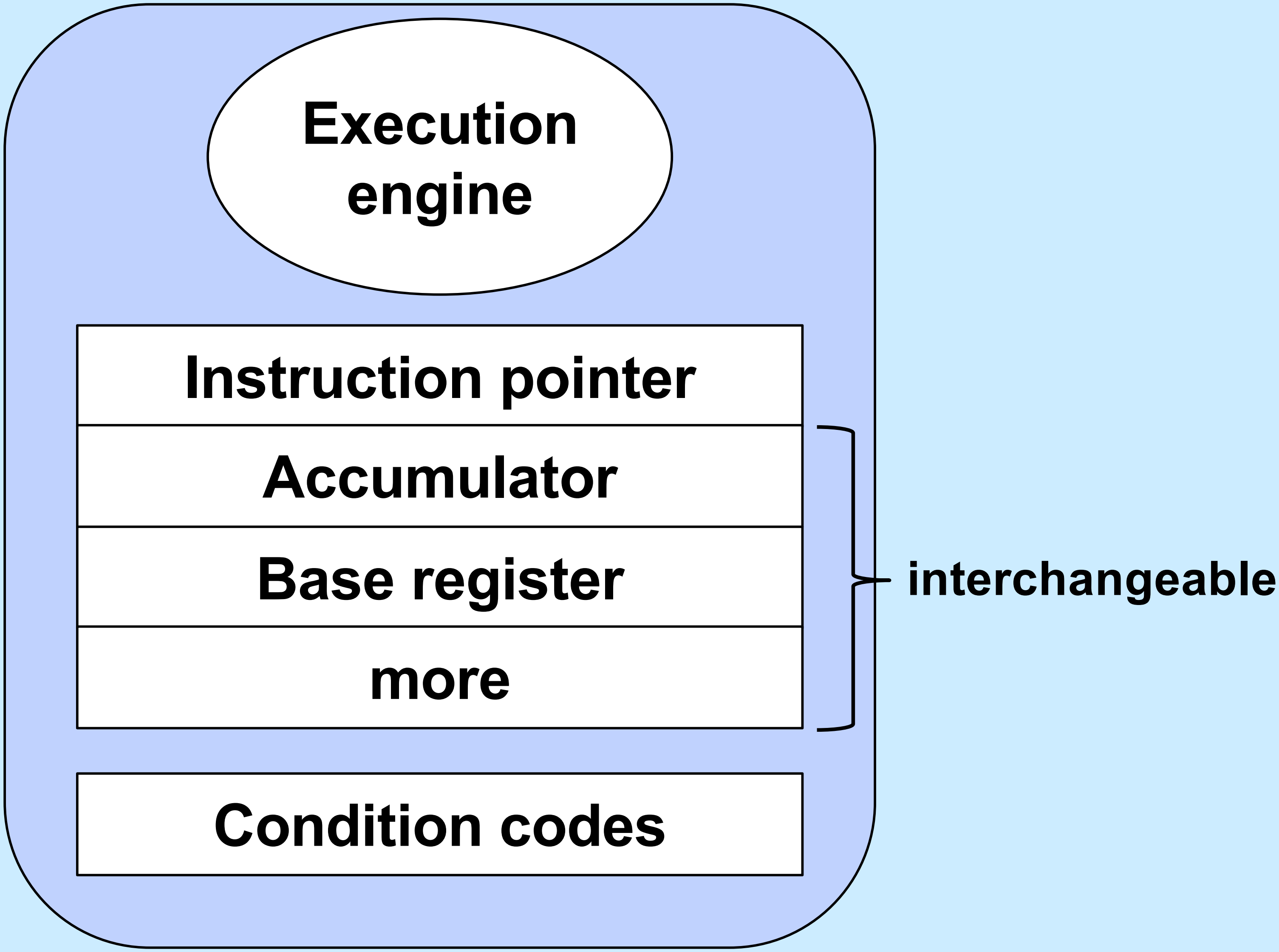
a) 9992
b) 9996
c) 10,004
d) 10,008

```
mov    1000,%acc
add    -8(%base),%acc
mul    -12(%base),%acc
mov    %acc,-16(%base)
```



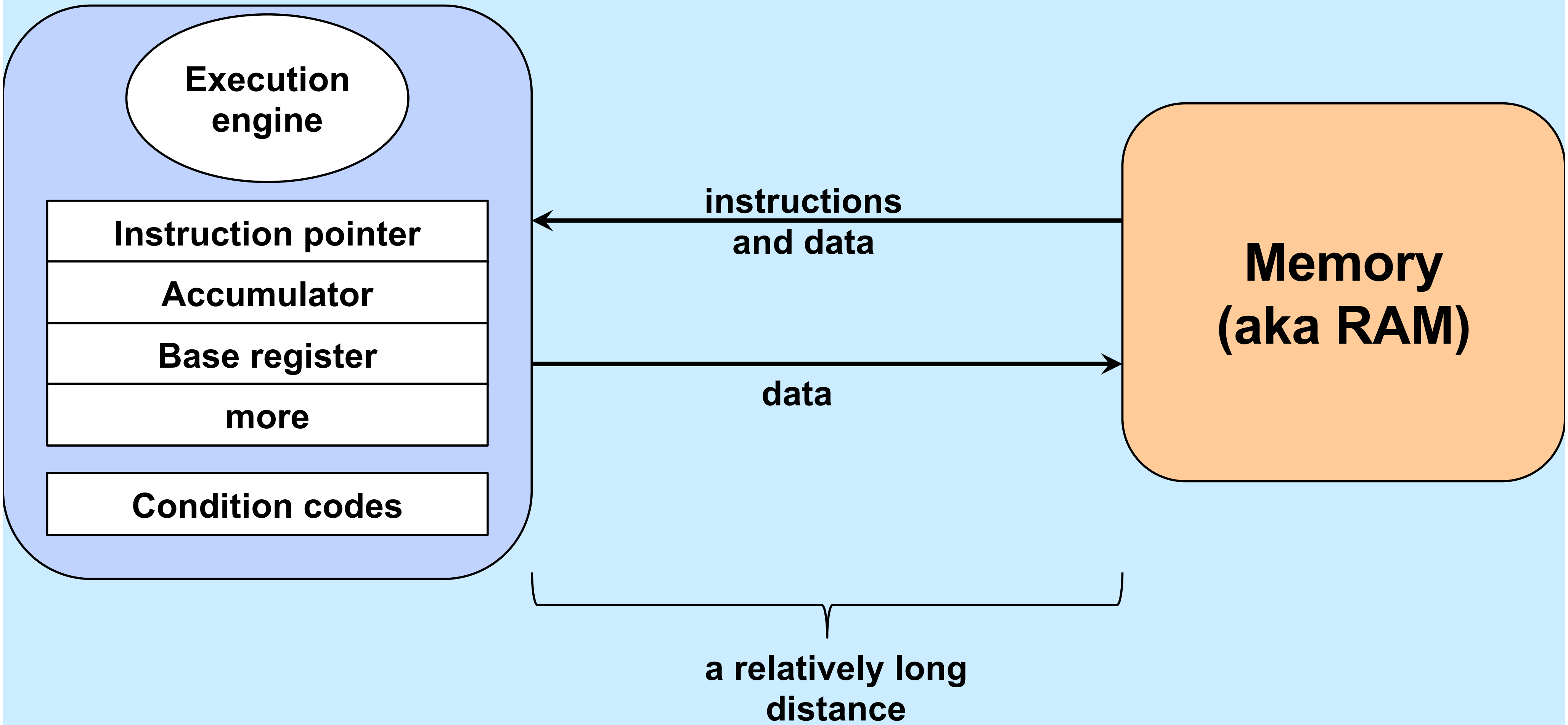
Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Registers



Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Registers vs. Memory



Copyright © 2019 Thomas W. Doeppner. All rights reserved.