Programming with Nios II

- Nios II is an embedded processor by Altera
- Other embedded processors
  - ARM, MIPS, 8051, 6812
    - 8051, 6812: tiny tasks (eg, keyboard controller)
    - ARM: in most smartphones
    - MIPS: similar to Nios, cost-sensitive
- Other processors
  - Pentium → "Sandybridge" Nehalem
  - Athlon
  - PowerPC, Cell
- Nios II is a "typical RISC processor"
  - easy to program, easy to understand, easy to build
  - high performance

Features

- 32 registers r0 to r31
  - each register stores a 32-bit number
  - but r0 is special – it always holds value 0
- Each instruction is always 32 bits in size
  - about 100 instructions, divided into 3 types:
    1. R type
    2. I type
    3. J type

1) R type - uses 3 registers

eg:
```
add  r1, r2, r3     ~  r1 = r2 + r3
sub  r2, r16, r15   ~  r2 = r16 - r15
mul  r3, r31, r6    ~  r3 = r31 * r6
div  r6, r2, r3     ~  r6 = r2 / r3
```

2) I type - uses 2 registers plus an immediate operand

eg:
```
addi r1, r2, 14
subi r2, r16, -4
muli r3, r3, 2
```

note: no divi instruction

"IMM16" a constant number 16 bits in size usually signed (depends on instruction)

3) J type - uses ∅ registers, only a 26-bit immediate operand

"IMM26"

eg:
```
call
jmpi
```

26 bit number is address of a subroutine; usually we write a text label here, not a number - examples later...

More Examples

R type	and	or	nor	xor
I type	andi	ori	—	xori

} for these logical instr., IMM16 is unsigned

eg: andi r3, r2, $F012

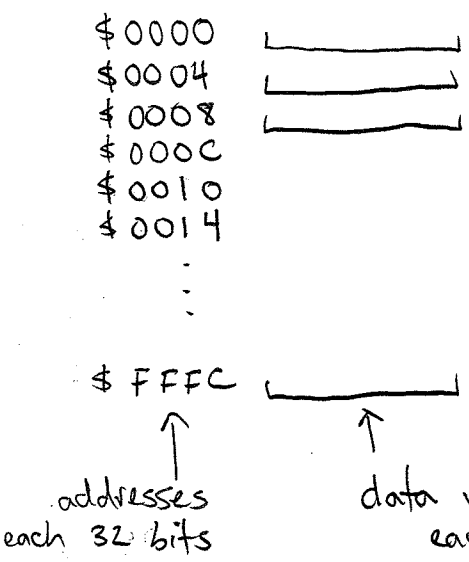
↳ "bitwise and"

```
r2     : a31 a30 ... a1 a0
& $F012 :  0   0  ...  1  0
-------------------------------
r3 →    :  0   0       a1 0
```

$a_{31} \cdot 0$ $a_{30} \cdot 0$ ($a_1 \cdot 1$) $a_0 \cdot 0$

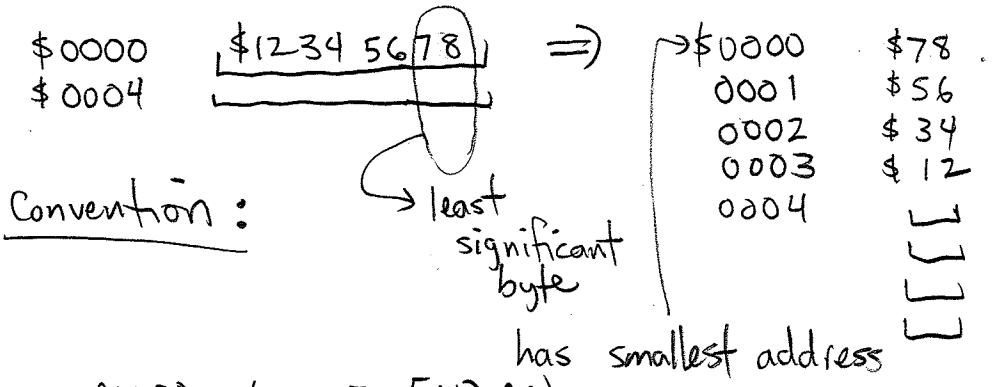
Memory

- Just registers aren't enough
- Need memory to hold - instructions, eg long programs
  - data, eg image pixels
- Organized as words (32 bits or 4 bytes)
- Each memory word has a location it is stored at, its address



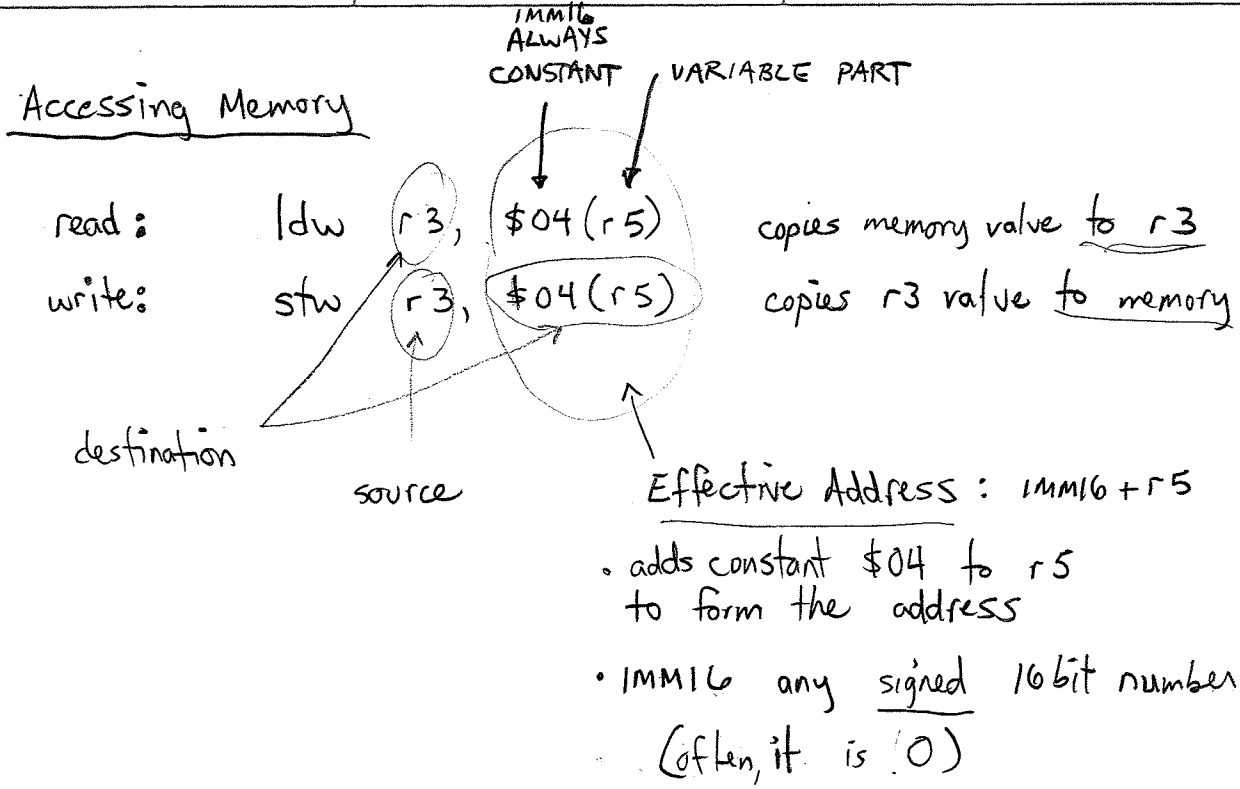
Notice: addresses are always a multiple of 4 for words

Why? because we some times need to access just the bytes instead, so we give each byte of memory a unique address



CALLED LITTLE ENDIAN

Accessing Memory



To access memory, we need to first form an address
The address is formed by adding a 16 bit constant to a register — above, it is $04 + r5