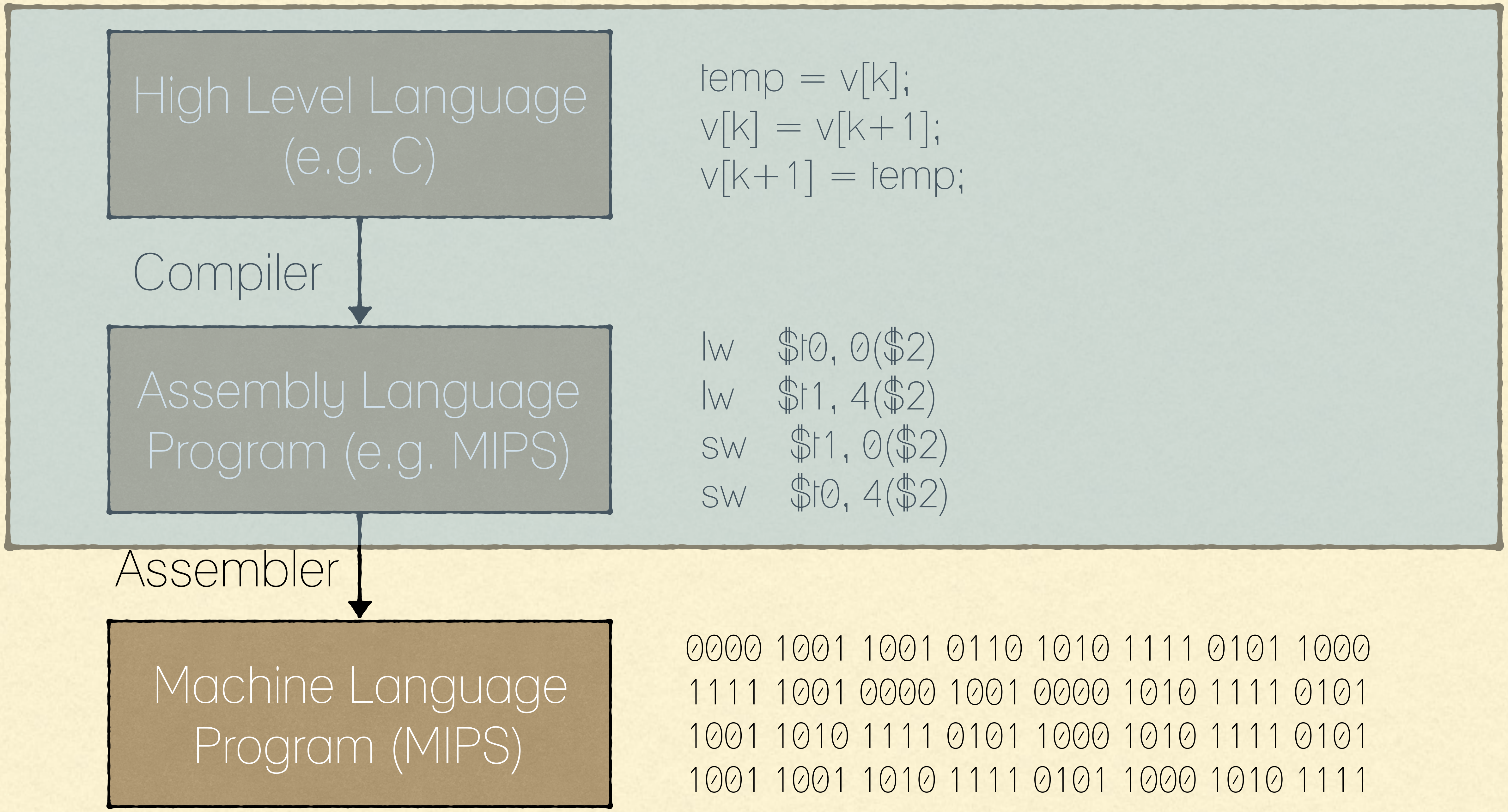
MIPS ASSEMBLY PROGRAMMING LANGUAGE VII & VIII

Ayman Hajja, PhD

LEVELS OF REPRESENTATION

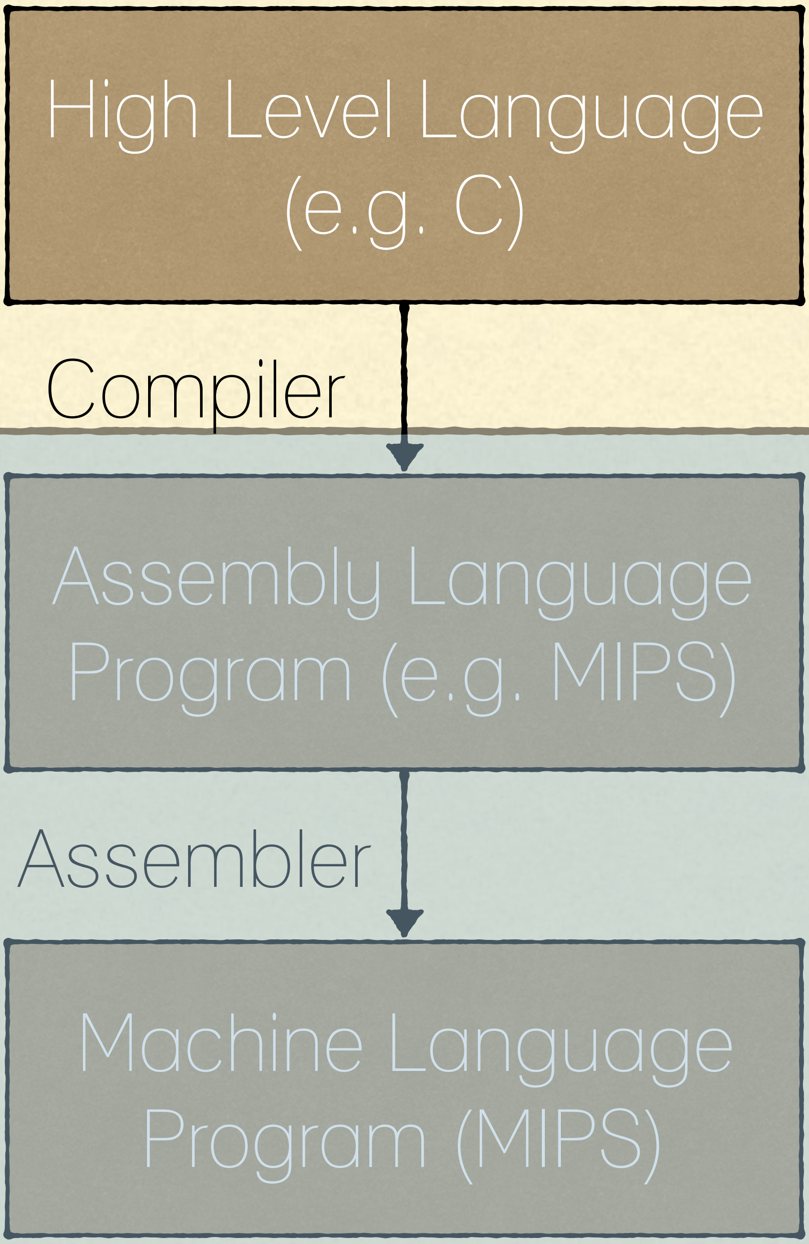

High Level Language (e.g. C)
Compiler
Assembly Language Program (e.g. MIPS)
Assembler
Machine Language Program (MIPS)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

```
0000 1001 1001 0110 1010 1111 0101 1000
1111 1001 0000 1001 0000 1010 1111 0101
1001 1010 1111 0101 1000 1010 1111 0101
1001 1001 1010 1111 0101 1000 1010 1111
```

LEVELS OF REPRESENTATION


High Level Language (e.g. C)
Compiler
Assembly Language Program (e.g. MIPS)
Assembler
Machine Language Program (MIPS)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

```
0000 1001 1001 0110 1010 1111 0101 1000
1111 1001 0000 1001 0000 1010 1111 0101
1001 1010 1111 0101 1000 1010 1111 0101
1001 1001 1010 1111 0101 1000 1010 1111
```

BINARY COMPATIBILITY

- Programs are distributed in binary form;
    - Programs bound to specific instruction set
- New machines want to run old programs ("binaries") as well as programs compiled to new instructions
    - Leads to "backward-compatible" instruction set evolving over time

INSTRUCTIONS AS NUMBERS

- Each instruction is a 32-bit sequence; we divide instructions into "fields"
- Each field tells processor something about instruction
- We could define different fields for each instruction, but MIPS/ RISC seeks simplicity, it defines 3 basic types of instructions format;
    - R-Format
    - I-Format
    - J-Format

INSTRUCTION FORMATS

- I-Format; used for instructions with immediates, 'lw' and 'sw' (offset counts as an immediate), and branches ('beq' and 'bne')
- J-Format; used for 'j' and 'jal'
- R-Format; used for all other instructions

R-FORMAT INSTRUCTIONS

- Define "fields" of the following number of bits each;
- 6 + 5 + 5 + 5 + 5 + 6 = 32

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- For simplicity, each field has a name;

opcode	rs	rt	rd	shamt	funct

R-FORMAT INSTRUCTIONS

- What do these field values tell us?
  - opcode; partially specifies what instruction it is
    - Note; This field is all 0s in all R-Format instructions
  - funct; combined with 'opcode', this number specifies what the instruction is

R-FORMAT INSTRUCTIONS

- More fields;
  - rs (Source Register); usually used to specify register containing first operand
  - rt (Target Register); usually used to specify register containing second operand (note that name is misleading)
  - rd (Destination Register); usually used to specify register which will receive result of computation

R-FORMAT INSTRUCTIONS

- Notes about register field;
    - Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0 to 31. Each of these fields refers to one of the 32 registers by number.

R-FORMAT INSTRUCTIONS

- Final field;
  - shamt; This field contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is pointless, which explains why this field is only 5 bits
  - This field is set to `0` in all but the shift instructions

R-FORMAT EXAMPLE

- MIPS Instruction;
  - add \$8, \$9, \$10 (add \$t0, \$t1, \$t2)
  - opcode = 0 (look up in table in book)
  - funct = 32 (look up in table in book)
  - rd = 8 (destination)
  - rs = 9 (first operand)
  - rt = 10 (second operand)
  - shamt = 0 (not a shift)

R-FORMAT EXAMPLE

- MIPS Instruction:
  - add \$8, \$9, \$10 (add \$t0, \$t1, \$t2)
  - Decimal number per field representation;

0	9	10	8	0	32

  - Binary number per field representation;

0000 00	01 001	0 1010	0100 0	000 00	10 0000

  - hex representation;   0x012A4020
  - Called a machine language instruction

I-FORMAT INSTRUCTIONS

- Define "fields" of the following number of bits each;
  - 6 + 5 + 5 + 16 = 32 bits

6 bits	5 bits	5 bits	16 bits

  - Again, each field has a name;

opcode	rs	rt	immediate

I-FORMAT EXAMPLE

- MIPS Instruction;
  - addi $21, $22, -50 (addi $s5, $s6, -50)
  - Decimal representation;

8	22	21	-50

  - Binary representation

0010 00	10 110	1 0101	1111 1111 1100 1110

  - hexadecimal representation; 0x22d5ffce

I-FORMAT INSTRUCTIONS

- The meaning of the fields;
  - opcode; same as before, except that, there's no function field, opcode uniquely identifies an instruction in I-format
  - rs; specifies the register operand (there's only one)
  - rt; specifies the register which receive result of computation (this is why it's called the target register "rt")

QUESTION

- Which instruction has same representation as the following:

0000 0010 0011 0001 1000 1000 0010 0010

A. add $zero, $zero, $zero
B. sub $s1, $s1, $s1
C. lw $zero, 0($zero)
D. addi $zero, $zero, 35
E. sub $zero, $zero, $zero

opcode	rs	rt	rd	shamt	funct

opcode	rs	rt	immediate

- 0: $zero, 8: $t0, 9: $t1, …15: $t7, 16: $s0, 17: $s1, …23: $s7
- add: opcode = 0, funct = 32
- sub: opcode = 0, funct = 34
- addi: opcode = 8
- lw: opcode = 35

QUESTION

- Which instruction has same representation as the following:

0000 0010 0011 0001 1000 1000 0010 0010

A. add $zero, $zero, $zero

[✓ B. sub $s1, $s1, $s1]

C. lw $zero, 0($zero)

D. addi $zero, $zero, 35

E. sub $zero, $zero, $zero

opcode	rs	rt	rd	shamt	funct

opcode	rs	rt	immediate

  - 0: $zero, 8: $t0, 9: $t1, …15: $t7, 16: $s0, 17: $s1, …23: $s7
  - add: opcode = 0, funct = 32
  - sub: opcode = 0, funct = 34
  - addi: opcode = 8
  - lw: opcode = 35

I-FORMAT INSTRUCTIONS

- The Immediate Field;
    - 16 bits can be used to represent immediate up to 2^{16} different values
- What do we do if we need more than 16 bits, for example, how would we addi $s0, $t0, 0x0ab0 f0a9?

I-FORMAT INSTRUCTIONS

- The Immediate Field;
  - 16 bits can be used to represent immediate up to 2^{16} different values
- What do we do if we need more than 16 bits, for example, how would we addi $s0, $t0, 0x0ab0 f0a9?

We first place the large immediate in a register, 'or' it with the lower half, then add it to to $t0 and place the result in $s0

DEALING WITH LARGE IMMEDIATES

- Example 01; addi $s0, $t0, 0x019f 23a2

DEALING WITH LARGE IMMEDIATES

- Example 01; addi $s0, $t0, 0x019f 23a2
    - lui $at, 0x019f

DEALING WITH LARGE IMMEDIATES

- Example 01; addi $s0, $t0, 0x019f 23a2
  - lui $at, 0x019f
  - ori $at, $at, 0x23a2

DEALING WITH LARGE IMMEDIATES

- Example 01; addi $s0, $t0, 0x019f 23a2
    - lui $at, 0x019f
    - ori $at, $at, 0x23a2
    - add $s0, $t0, $at

DEALING WITH LARGE IMMEDIATES

- Example 01; addi $s0, $t0, 0x019f 23a2
  - lui $at, 0x019f
  - ori $at, $at, 0x23a2
  - add $s0, $t0, $at

Reserved temporary register for assembly

This register contains our 32-bit immediate

DEALING WITH LARGE IMMEDIATES

- Example 02; lw $s0, 0x019f 23a2($s1)

DEALING WITH LARGE IMMEDIATES

- Example 02; lw \$s0, 0x019f 23a2(\$s1)

Let's put 0x019f 23a2 in a register; add that register to \$s1 and store the result in another register (say \$t0), and finally let's use \$t0 (with offset 0) to load \$s0 into that address

DEALING WITH LARGE IMMEDIATES

- Example 02; lw $s0, 0x019f 23a2($s1)
  - addi $t0, $s1, 0x019f 23a2

DEALING WITH LARGE IMMEDIATES

- Example 02; lw $s0, 0x019f 23a2($s1)
    - addi $t0, $s1, 0x019f 23a2
        - lui $at, 0x019f
        - ori $at, $at, 0x23a2
        - add $t0, $s1, $at

This register contains our 32-bit immediate

DEALING WITH LARGE IMMEDIATES

- Example 02; lw $s0, 0x019f 23a2($s1)
    - addi $t0, $s1, 0x019f 23a2
        - lui $at, 0x019f
        - ori $at, $at, 0x23a2
        - add $t0, $s1, $at
    - lw $s0, 0($t0)

This register contains our 32-bit immediate

Now we load the value in $s0 into address $t0, which contains the 32-bit immediate + the value in $s1

NATURAL ALIGNMENT

- Data (and instructions) must be naturally aligned when stored in the RAM.
    - This means that if our data point that we want to store to the RAM, say an integer, is 4 bytes long, then the address of the first byte (out of the four) must be a multiple of 4 (the size of our data point)

NATURAL ALIGNMENT

...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
19	18	17	16
15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	0

Say each cell is a byte; each cell/ byte is assigned a unique address

For example, you may think that if we store an int (4 bytes) at address 9, then the entire int would be stored in address 9, 10, 11, and 12 (shown on the left). However, since 9 is not a multiple of 4 (size of int), storing an int at address 9 won't be permitted by the CPU in the first place.

NATURAL ALIGNMENT

- Examples of valid instructions;
    - sw $t0, 16($zero) # Valid since 16 is a multiple of 4
    - sh $t0, 18($zero) # sh stands for store half (18 is multiple of 2)
    - lh $t0, 22($zero) # lh stands for load half (22 is multiple of 2)
    - sb $t0, 17($zero) # sb stands for store byte (17 is a multiple of 1)
    - lb $t0, 13($zero) # lb stands for load byte (any number is a multiple of 1)
- Examples of invalid instructions (will cause <u>alignment fault</u>);
    - sw $t0, 18($zero) # 18 is not a multiple of 4
    - lh $t0, 17($zero) # 17 is not a multiple of 2
    - sh $to, 19($zero)# 19 is not a multiple of 2
    - lw $t0, 26($zero) # 26 is not a multiple of 4

BRANCHING INSTRUCTIONS

- beq and bne
    - Need to specify a target address if branch taken
    - Also specify two registers to compare
    - Use I-Format

opcode	rs	rt	immediate

- opcode specifies beq or bne
- rs and rt specify registers

BRANCHING INSTRUCTIONS

- Branches typically used in loops (if/else, while, for)
  - Loops are generally small
  - Function calls and unconditional jumps handled with jump instruction (J-Format)
- Recall; Instructions stored in a localized area of memory (Code/ Text)
  - Largest branch distance limited by size of code
  - Address of current instruction stored in Program Counter

PC-RELATIVE ADDRESSING

- PC-Relative Addressing; Use the immediate fields as a two's compliment offset to PC
    - Branches generally change the PC by a small amount
    - Can specify 2^{16} - 1 addresses from the PC

PC-RELATIVE ADDRESSING

- PC-Relative Addressing; Use the immediate fields as a two's compliment offset to PC
  - Branches generally change the PC by a small amount
  - Can specify 2^{16} - 1 addresses from the PC

Can we do better?

BRANCHING REACH

- Recall; MIPS uses 32-bit addresses
  - Memory is byte-addressable
- Instructions are word-aligned
  - Address is always multiple of 4 (in bytes), meaning it ends with `0b00` in binary
  - Number of bytes to add to the PC will always be multiple of 4
- Immediate specifies words instead of bytes
  - Can now reach up to 2^{16} instructions (2^{18} bytes around PC)

BRANCH CALCULATION

- If we don't take the branch:
    - Next instruction will be PC + 4
- If we do take the branch;
    - Next instruction will be (PC + 4) + (immediate*4)
- Observations;
    - Immediate is number of instructions to jump (remember, specifies word) either forward (positive) or backwards (negative)

BRANCH EXAMPLE

- MIPS Code;

```
Loop:
1000  beq $9, $zero, End
1004  add  $8, $8, $10
1008  addi $9, $9, -1
1012  j Loop
End:
```

- I-Format fields;
  - opcode = 4   (look up on Green Sheet)
  - rs = 9   (first operand)
  - rt = 0   (second operand)
  - What's the immediate value (End)?

BRANCH EXAMPLE

- MIPS Code;

```
Loop:
1000  beq $9, $zero, End
1004  add  $8, $8, $10
1008  addi $9, $9, -1
1012  j Loop
End:
```

4	9	0	3

- I-Format fields;
  - opcode = 4 (look up on Green Sheet)
  - rs = 9 (first operand)
  - rt = 0 (second operand)
  - What's the immediate value (End)? 3