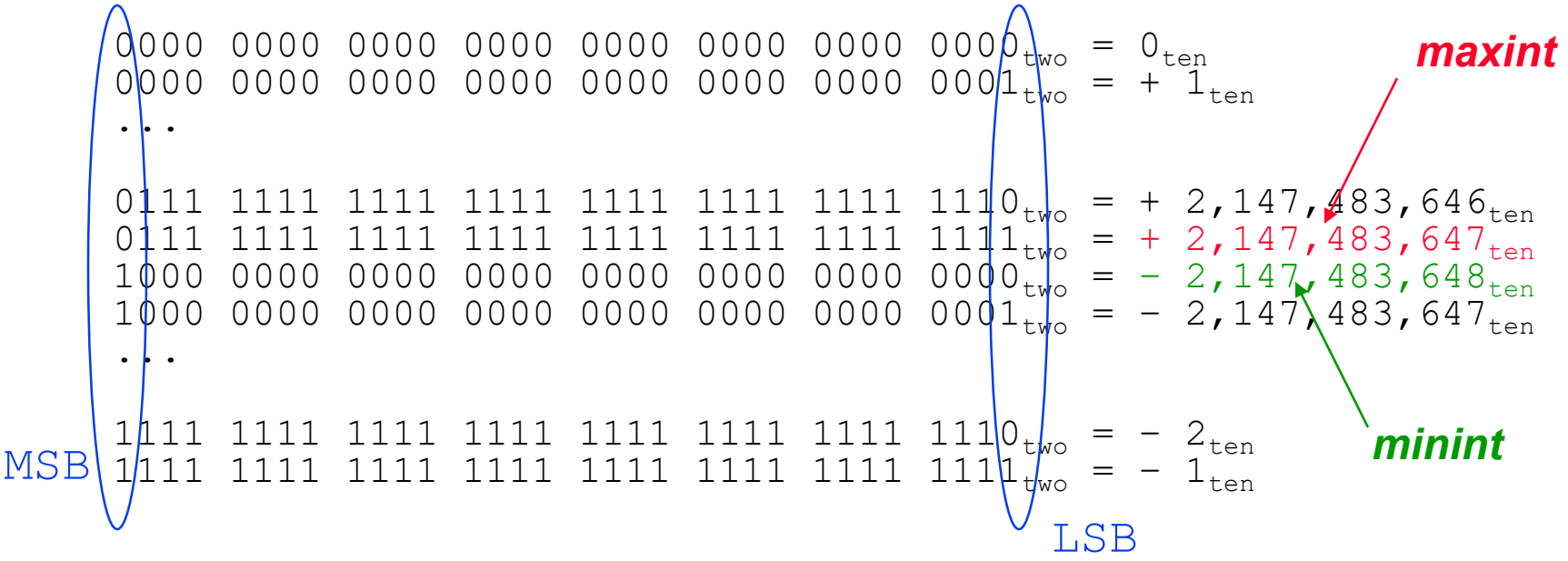
Arithmetic for Computers

[Adapted from Computer Organization and Design, Patterson & Hennessy, courtesy for Mary Jane Irwin]

Number Representations

- 32-bit signed numbers (2's complement):



- Converting <32-bit values into 32-bit values
  - copy the most significant bit (the sign bit) into the "empty" bits

```
0010   -> 0000 0010
1010   -> 1111 1010
```

  - sign extend versus zero extend (`lb` vs. `lbu`)

Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA

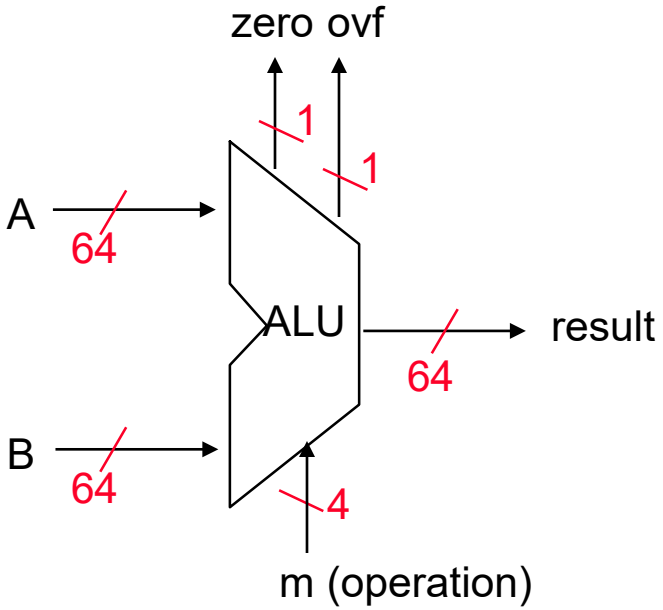
  `add, addi, addw, addiw`

  `sub, subw`

  `mul, mulw, div, divw`

  `and, andi, or, ori, xor, xori`

  `beq, bne, blt, bltu, bge, bgeu, sll, slli`



- With special handling for
  - sign extend – `add, sub, mul, div`
  - zero extend – `andi, ori, xori`
  - overflow detection – `add, addi, sub`

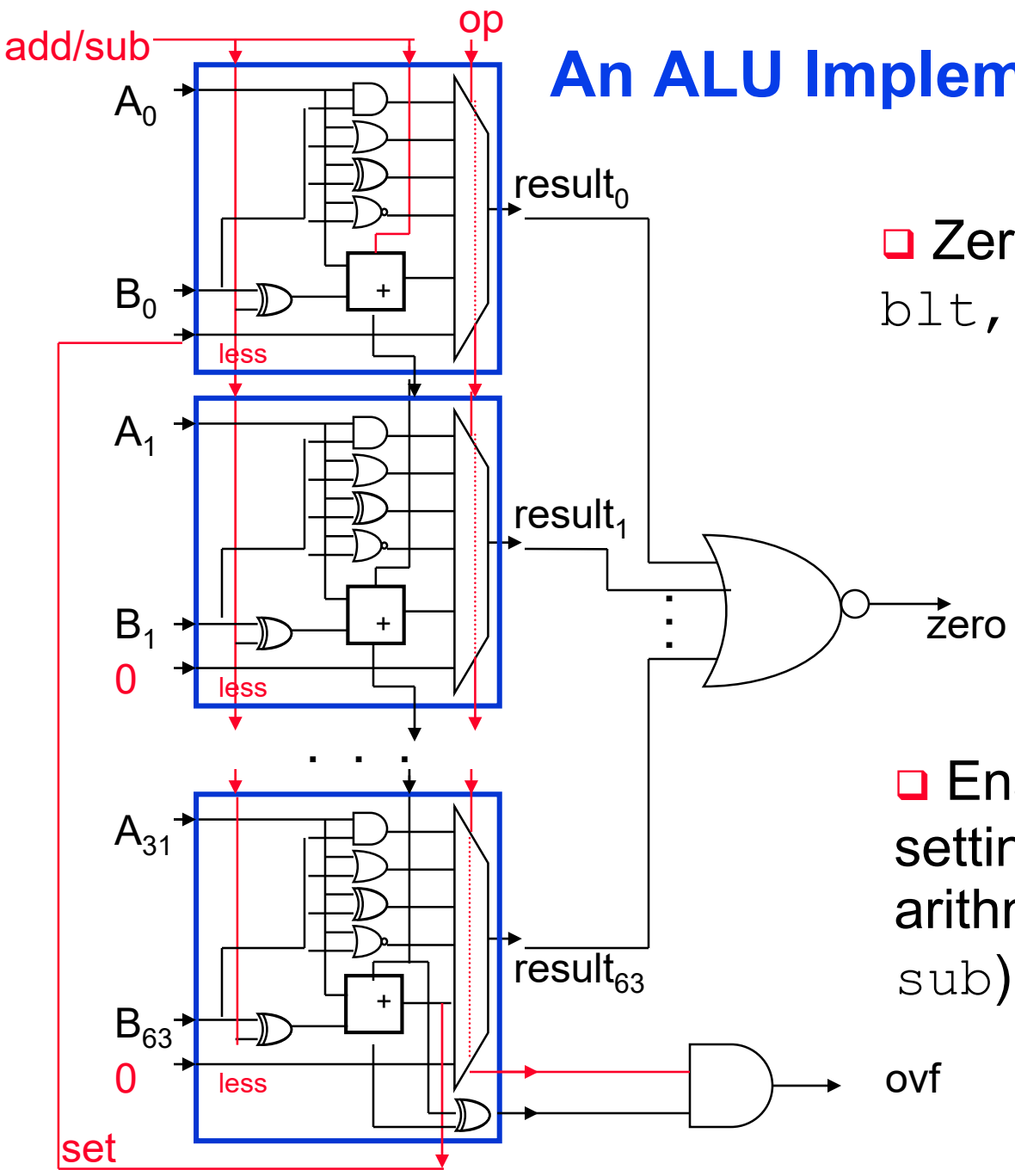
Dealing with Overflow

- Overflow occurs when the result of an operation cannot be represented in 64-bits, i.e., when the sign bit contains a value bit of the result and not the proper sign bit
  - When adding operands with different signs or when subtracting operands with the same sign, overflow can *never* occur

Operation	Operand A	Operand B	Result indicating overflow
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

- RISC-V signals overflow with an exception (aka interrupt) – an unscheduled procedure call where the EPC contains the address of the instruction that caused the exception

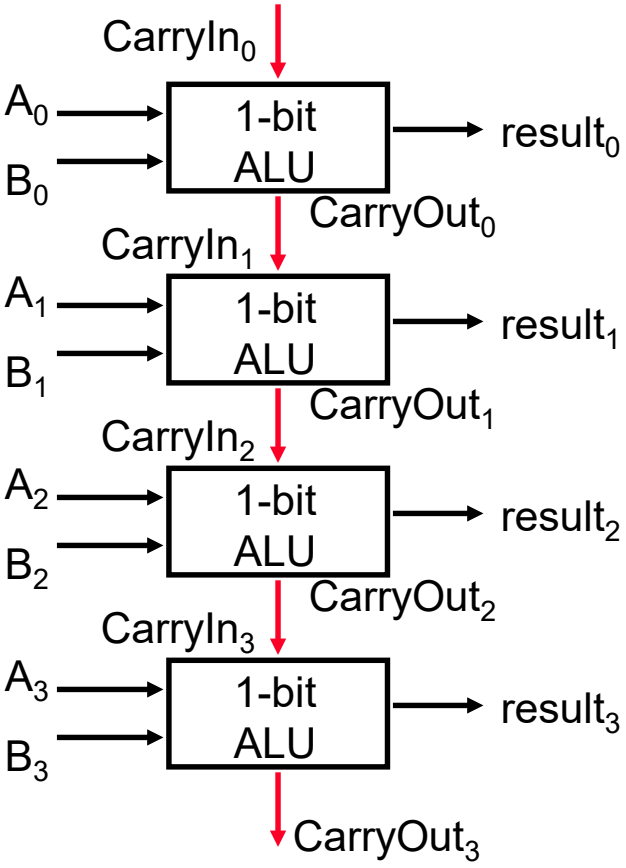
An ALU Implementation



- Zero detect (`beq, bne, blt, bge`)
- Enable overflow bit setting for signed arithmetic (`add, addi, sub`)

But What about Performance?

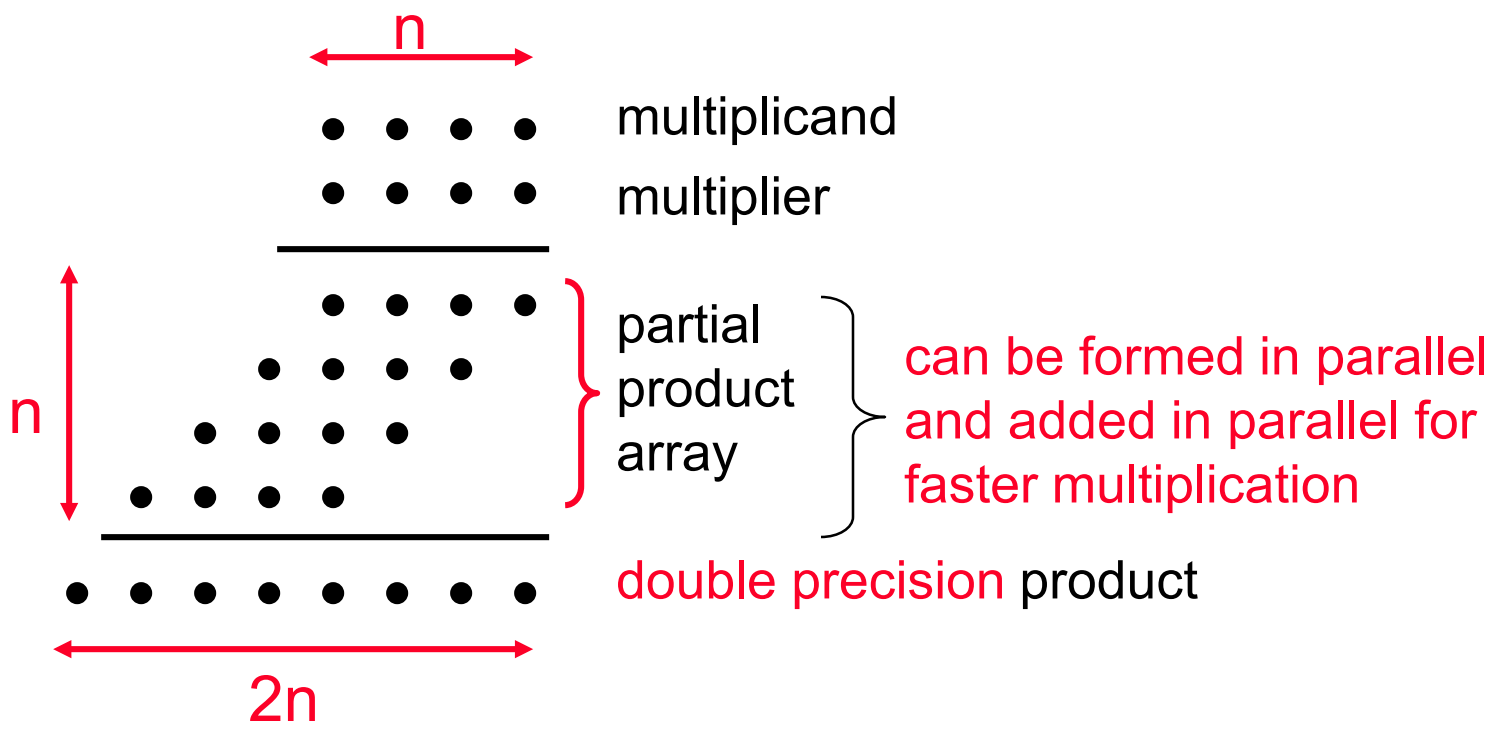
- Critical path of n-bit ripple-carry adder is n*CP



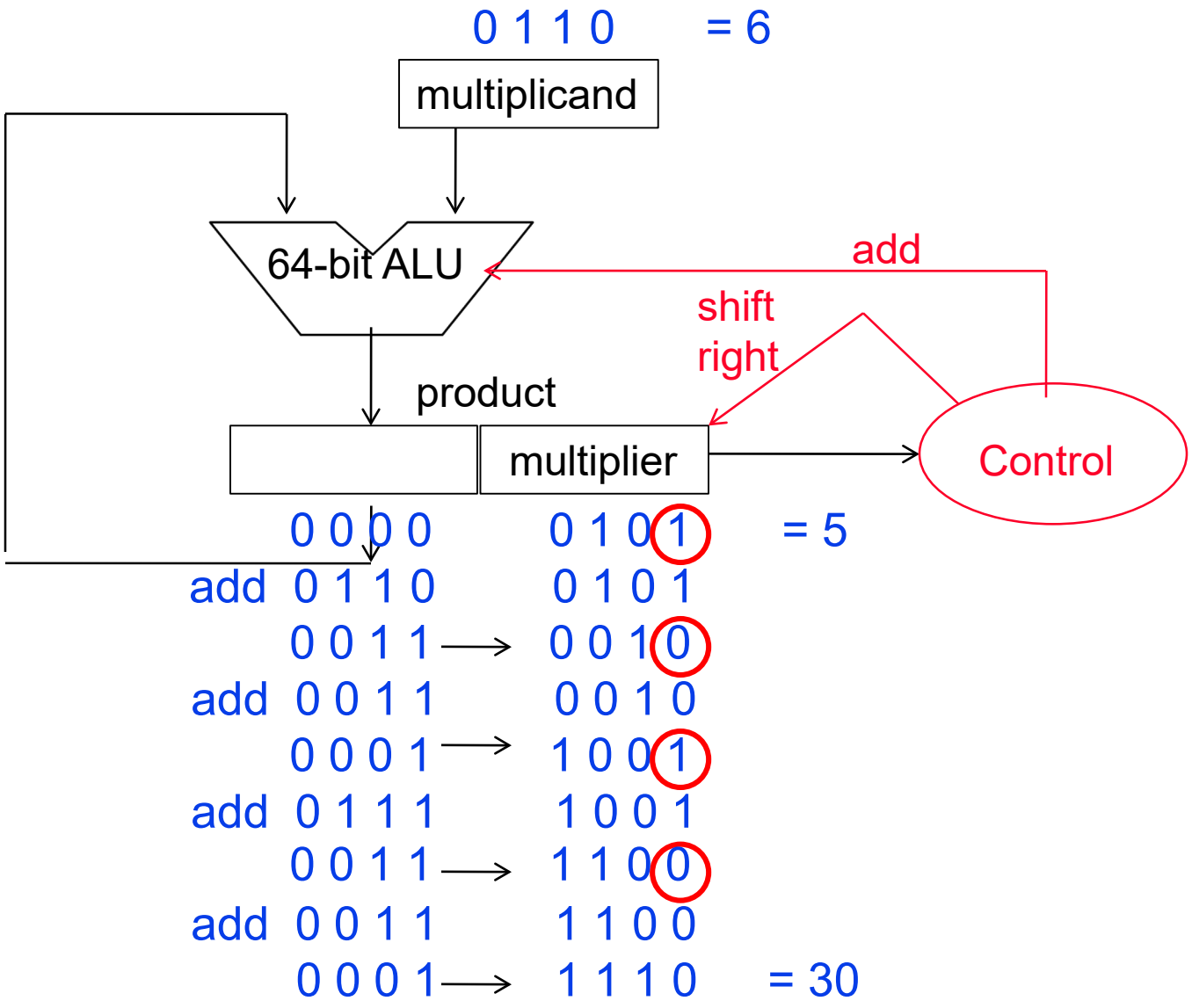
- Design trick – throw hardware at it (Carry Lookahead)

Multiply

- Binary multiplication is just a *bunch* of right shifts and adds



Add and Right Shift Multiplier Hardware



RISC-V Multiplication

- Four multiply instructions:
  - mul: multiply
    - Gives the lower 64 bits of the product
  - mulh: multiply high
    - Gives the upper 64 bits of the product, assuming the operands are signed
  - mulhu: multiply high unsigned
    - Gives the upper 64 bits of the product, assuming the operands are unsigned
  - mulhsu: multiply high signed/unsigned
    - Gives the upper 64 bits of the product, assuming one operand is signed and the other unsigned
  - Use mulh result to check for 64-bit overflow

MIPS Multiply Instruction

- Multiply (`mult` and `multu`) produces a double precision product

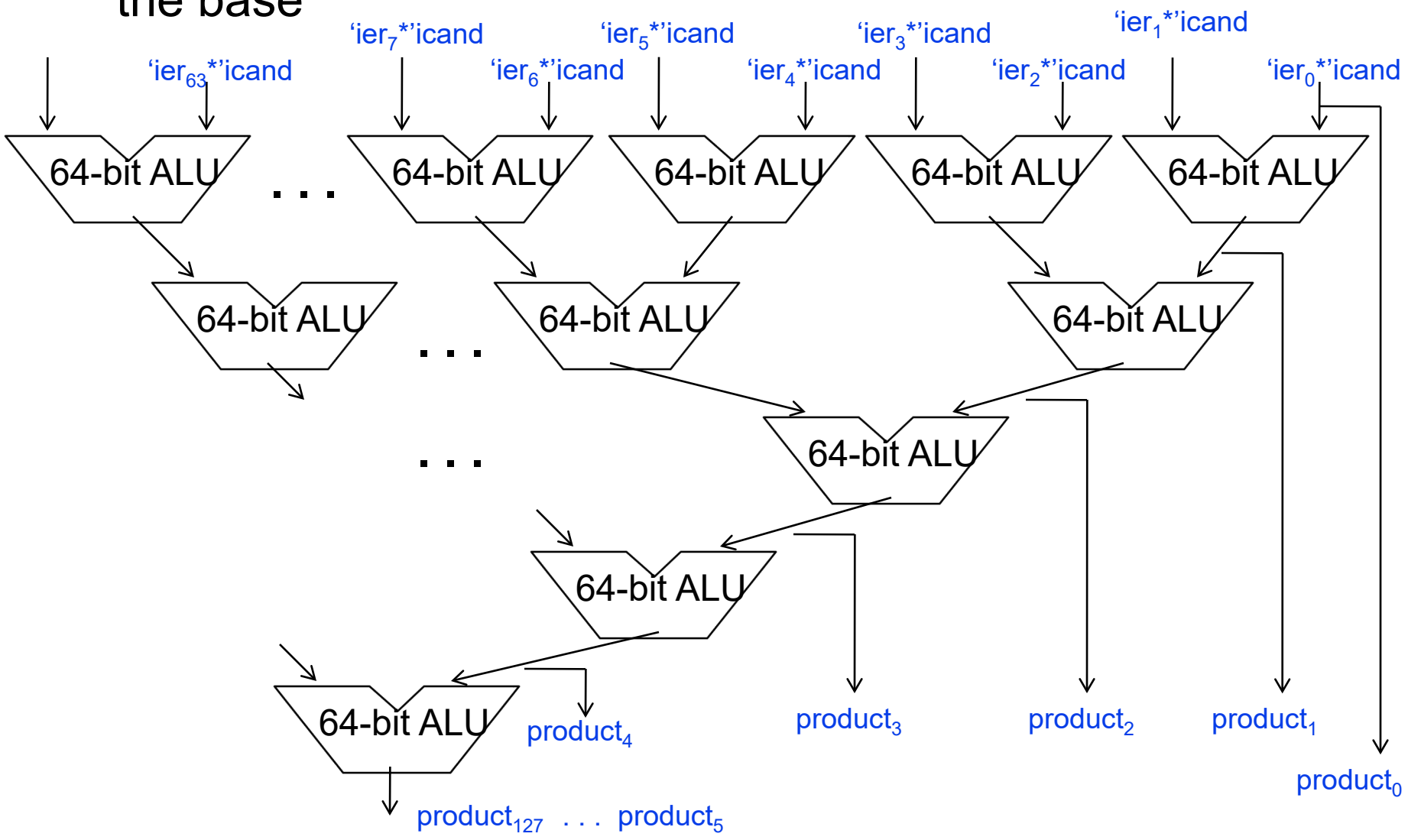
```
mult    $s0, $s1        # hi||lo = $s0 * $s1
```

  - Low-order word of the product is left in processor register `lo` and the high-order word is left in register `hi`
  - Instructions `mfhi rd` and `mflo rd` are provided to move the product to (user accessible) registers in the register file

- Multiplies are usually done by fast, dedicated hardware and are much more complex (and slower) than adders

Fast Multiplication Hardware

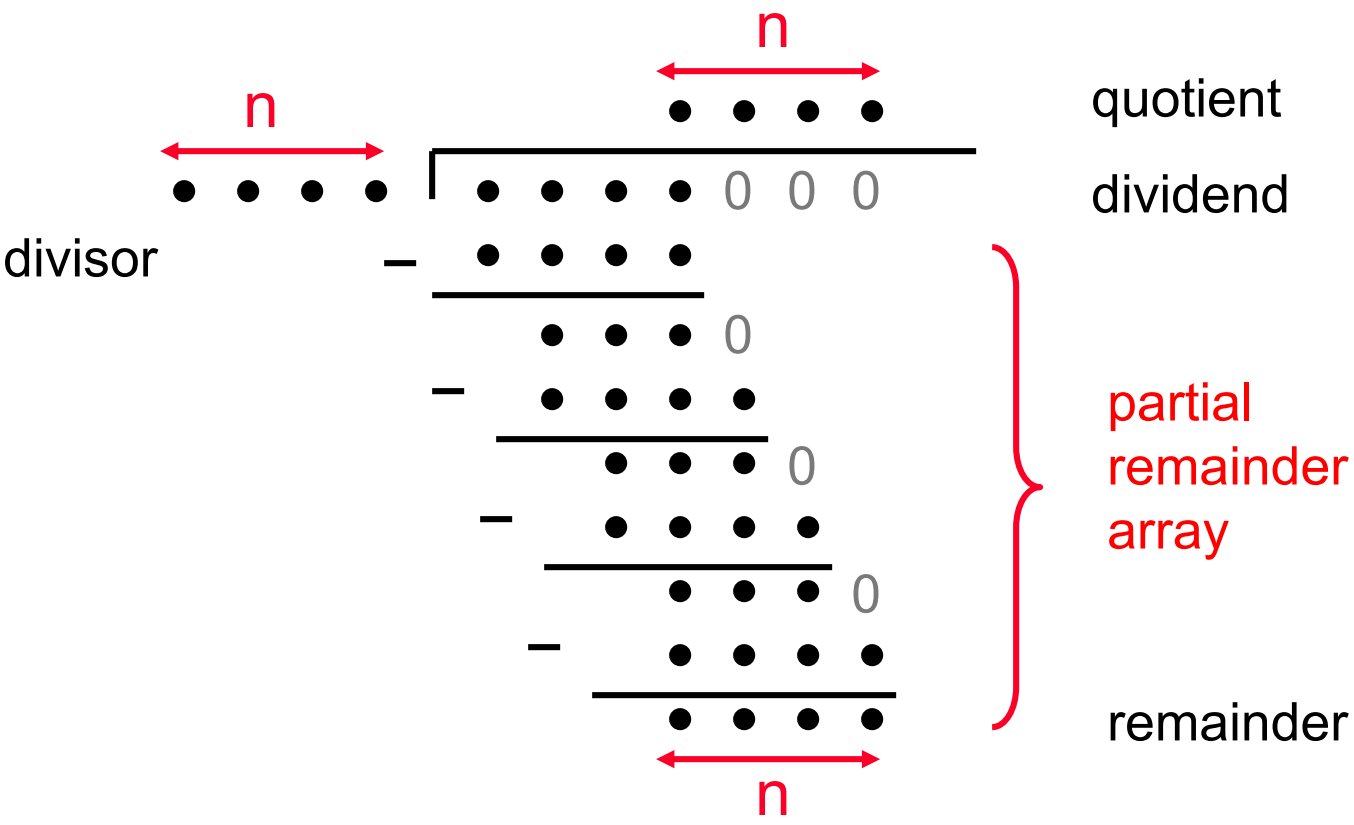
- Can build a faster multiplier by using a parallel tree of adders with one 64-bit adder for each bit of the multiplier at the base



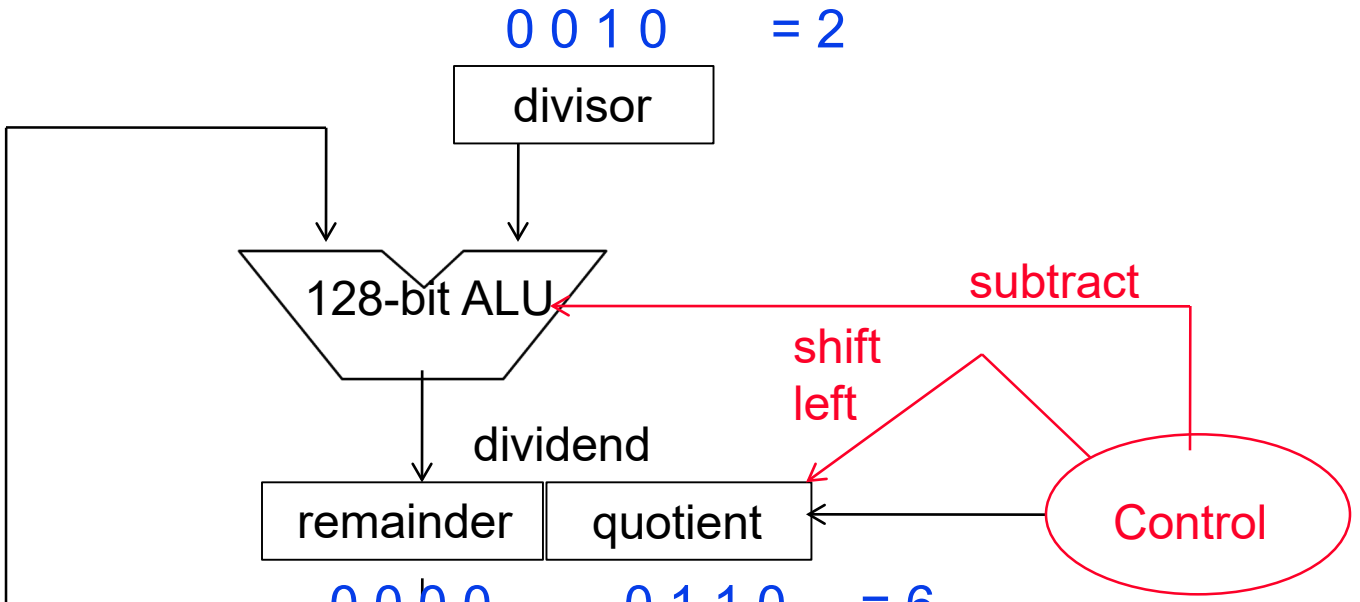
Division

- Division is just a *bunch* of quotient digit guesses and left shifts and subtracts

dividend = quotient x divisor + remainder



Left Shift and Subtract Division Hardware



	0 0 0 0	0 1 1 0	= 6
	0 0 0 0 ←	1 1 0 0	
sub	1 1 1 0	1 1 0 0	rem neg, so ‘ient bit = 0
	0 0 0 0	1 1 0 0	restore remainder
	0 0 0 1 ←	1 0 0 0	
sub	1 1 1 1	1 1 0 0	rem neg, so ‘ient bit = 0
	0 0 0 1	1 0 0 0	restore remainder
	0 0 1 1 ←	0 0 0 0	
sub	0 0 0 1	0 0 0 1	rem pos, so ‘ient bit = 1
	0 0 1 0 ←	0 0 1 0	
sub	0 0 0 0	0 0 1 1	rem pos, so ‘ient bit = 1

= 3 with 0 remainder

RISC-V Division Instructions

- Four instructions:
  - div, rem: signed divide, remainder (modulo operation)
  - divu, remu: unsigned divide, remainder (modulo operation)

- Overflow and division-by-zero don’t produce errors
  - Just return defined results
  - Faster for the common case of no error

Representing Big (and Small) Numbers

- What if we want to encode the approx. age of the earth?

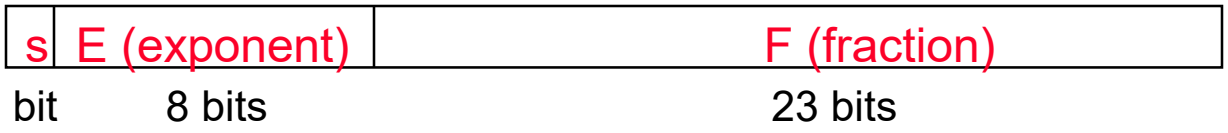
  4,600,000,000 or 4.6×10^9

  or the weight in kg of one a.m.u. (atomic mass unit)

  0.0000000000000000000000000166 or 1.6×10^{-27}

  There is no way we can encode either of the above in a 32-bit integer.

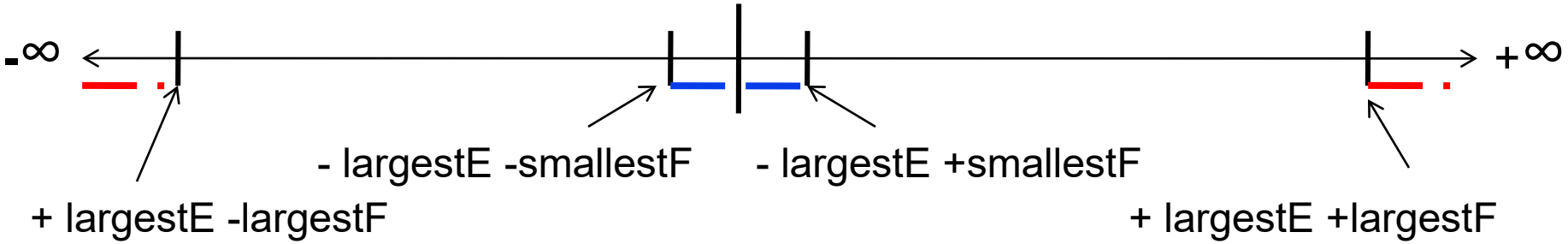
- Floating point representation $(-1)^{sign} \times F \times 2^E$
  - Still have to fit everything in 32 bits (single precision)



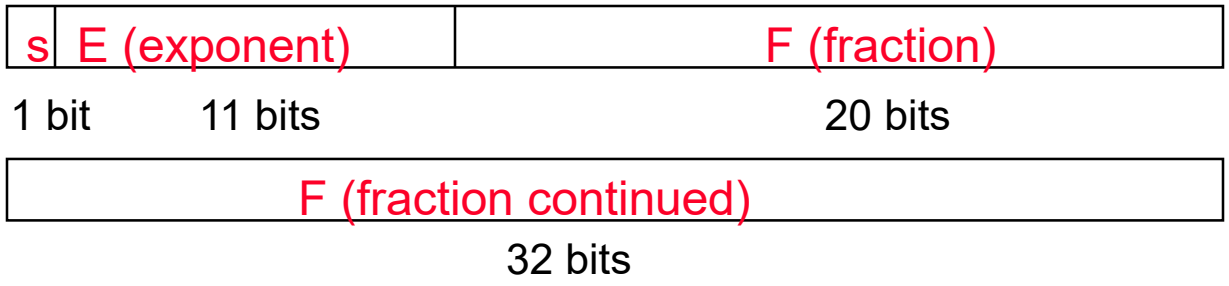
  - The base (2, *not* 10) is hardwired in the design of the FPALU
  - More bits in the fraction (F) or the exponent (E) is a trade-off between precision (accuracy of the number) and range (size of the number)

Exception Events in Floating Point

- Overflow (floating point) happens when a positive exponent becomes too large to fit in the exponent field
- Underflow (floating point) happens when a negative exponent becomes too large to fit in the exponent field



- One way to reduce the chance of underflow or overflow is to offer another format that has a larger exponent field
  - Double precision – takes two words



IEEE 754 FP Standard

- Most (all?) computers these days conform to the IEEE 754 floating point standard $(-1)^{sign} \times (1+F) \times 2^{E-bias}$
  - Formats for both single and double precision
  - F is stored in normalized format where the msb in F is 1 (so there is no need to store it!) – called the hidden bit
  - To simplify sorting FP numbers, E comes before F in the word and E is represented in excess (biased) notation where the bias is -127 (-1023 for double precision) so the most negative is 00000001 = $2^{1-127} = 2^{-126}$ and the most positive is 11111110 = $2^{254-127} = 2^{+127}$

- Examples (in normalized format)
  - Smallest+: 0 00000001 1.00000000000000000000000 = $1 \times 2^{1-127}$
  - Zero: 0 00000000 00000000000000000000000 = true 0
  - Largest+: 0 11111110 1.11111111111111111111111 = $2-2^{-23} \times 2^{254-127}$
  - $1.0_2 \times 2^{-1}$ = 0 01111110 1.00000000000000000000000
  - $0.75_{10} \times 2^4$ = 0 10000010 1.10000000000000000000000

IEEE 754 FP Standard Encoding

- Special encodings are used to represent unusual events
  - ± infinity for division by zero
  - NAN (not a number) for the results of invalid operations such as 0/0
  - True zero is the bit string all zero

Single Precision		Double Precision		Object Represented
E (8)	F (23)	E (11)	F (52)	
0000 0000	0	0000 … 0000	0	true zero (0)
0000 0000	nonzero	0000 … 0000	nonzero	± denormalized number
1-254	anything	1-2046	anything	± floating point number
255	0	2047	-0	± infinity
255	nonzero	2047	nonzero	not a number (NaN)

Support for Accurate Arithmetic

- IEEE 754 FP rounding modes
  - Always round up (toward +∞)
  - Always round down (toward -∞)
  - Truncate
  - Round to nearest even (when the Guard || Round || Sticky are 100) – always creates a 0 in the least significant (kept) bit of F
- Rounding (except for truncation) requires the hardware to include extra F bits during calculations
  - Guard bit – used to provide one F bit when shifting left to normalize a result (e.g., when normalizing F after division or subtraction)
  - Round bit – used to improve rounding accuracy
  - Sticky bit – used to support Round to nearest even; is set to a 1 whenever a 1 bit shifts (right) through it (e.g., when aligning F during addition/subtraction)

F = 1 . xxxxxxxxxxxxxxxxxxxxxxx G R S

Floating Point Addition

- Addition (and subtraction)

$$(\pm F1 \times 2^{E1}) + (\pm F2 \times 2^{E2}) = \pm F3 \times 2^{E3}$$

  - Step 0: Restore the hidden bit in F1 and in F2
  - Step 1: Align fractions by right shifting F2 by E1 - E2 positions (assuming E1 ≥ E2) keeping track of (three of) the bits shifted out in G R and S
  - Step 2: Add the resulting F2 to F1 to form F3
  - Step 3: Normalize F3 (so it is in the form 1.XXXXX …)
    - If F1 and F2 have the same sign → F3 ∈[1,4) → 1 bit right shift F3 and increment E3 (check for overflow)
    - If F1 and F2 have different signs → F3 may require *many* left shifts each time decrementing E3 (check for underflow)
  - Step 4: Round F3 and possibly normalize F3 again
  - Step 5: Rehide the most significant bit of F3 before storing the result

Floating Point Addition Example

- Add

$$(0.5 = 1.0000 \times 2^{-1}) + (-0.4375 = -1.1100 \times 2^{-2})$$

  - Step 0: Hidden bits restored in the representation above
  - Step 1: Shift significand with the smaller exponent (1.1100) right until its exponent matches the larger exponent (so once)
  - Step 2: Add significands

    $1.0000 + (-0.111) = 1.0000 - 0.111 = 0.001$

  - Step 3: Normalize the sum, checking for exponent over/underflow

    $0.001 \times 2^{-1} = 0.010 \times 2^{-2} = .. = 1.000 \times 2^{-4}$

  - Step 4: The sum is already rounded, so we're done
  - Step 5: Rehide the hidden bit before storing

Floating Point Multiplication

- Multiplication

$$(\pm F1 \times 2^{E1}) \times (\pm F2 \times 2^{E2}) = \pm F3 \times 2^{E3}$$

  - Step 0: Restore the hidden bit in F1 and in F2
  - Step 1: Add the two (biased) exponents and subtract the bias from the sum, so E1 + E2 – 127 = E3

    also determine the sign of the product (which depends on the sign of the operands (most significant bits))
  - Step 2: Multiply F1 by F2 to form a double precision F3
  - Step 3: Normalize F3 (so it is in the form 1.XXXXX …)
    - Since F1 and F2 come in normalized $\rightarrow$ F3 $\in$[1,4) $\rightarrow$ 1 bit right shift F3 and increment E3
    - Check for overflow/underflow
  - Step 4: Round F3 and possibly normalize F3 again
  - Step 5: Rehide the most significant bit of F3 before storing the result

Floating Point Multiplication Example

- Multiply

$$(0.5 = 1.0000 \times 2^{-1}) \times (-0.4375 = -1.1100 \times 2^{-2})$$

  - Step 0: Hidden bits restored in the representation above
  - Step 1: Add the exponents (not in bias would be -1 + (-2) = -3 and in bias would be (-1+127) + (-2+127) – 127 = (-1 -2) + (127+127-127) = -3 + 127 = 124
  - Step 2: Multiply the significands
    1.0000 x 1.110 = 1.110000
  - Step 3: Normalized the product, checking for exp over/underflow
    1.110000×2^{-3} is already normalized
  - Step 4: The product is already rounded, so we're done
  - Step 5: Rehide the hidden bit before storing

FP Instructions in RISC-V

- Separate FP registers: f0, …, f31
  - double-precision
  - single-precision values stored in the lower 32 bits
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - flw, fld
  - fsw, fsd

FP Instructions in RISC-V

- Single-precision arithmetic
  - `fadd.s, fsub.s, fmul.s, fdiv.s, fsqrt.s`
    - e.g., `fadds.s f2, f4, f6`
- Double-precision arithmetic
  - `fadd.d, fsub.d, fmul.d, fdiv.d, fsqrt.d`
    - e.g., `fadd.d f2, f4, f6`
- Single- and double-precision comparison
  - `feq.s, flt.s, fle.s`
  - `feq.d, flt.d, fle.d`
  - Result is 0 or 1 in integer destination register
    - Use beq, bne to branch on comparison result
- Branch on FP condition code true or false
  - `B.cond`

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow