Floating-point to Fixed-point Conversion for Efficient Implementation of Digital Signal Processing Programs

Wonyong Sung

School of Electrical Engineering
Seoul National University

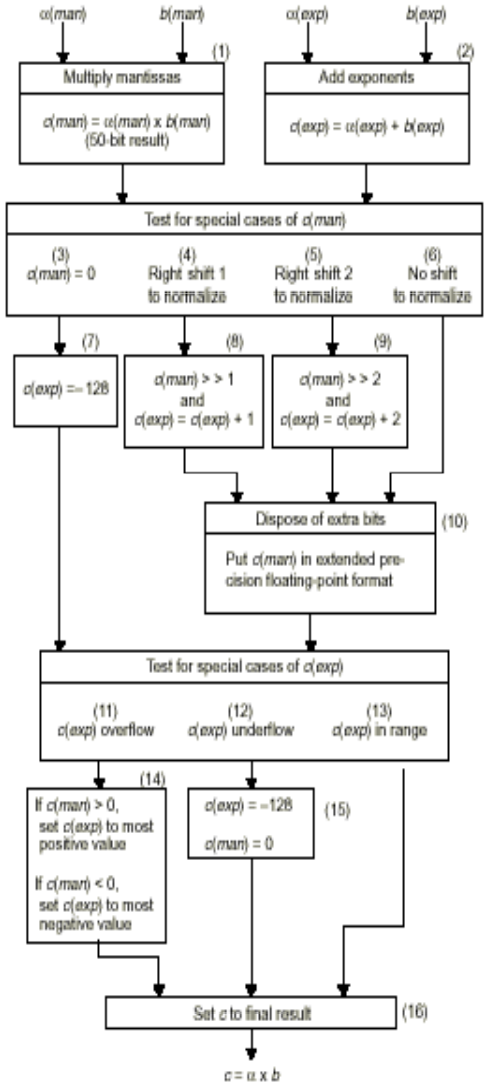
Version 2003. 7. 18

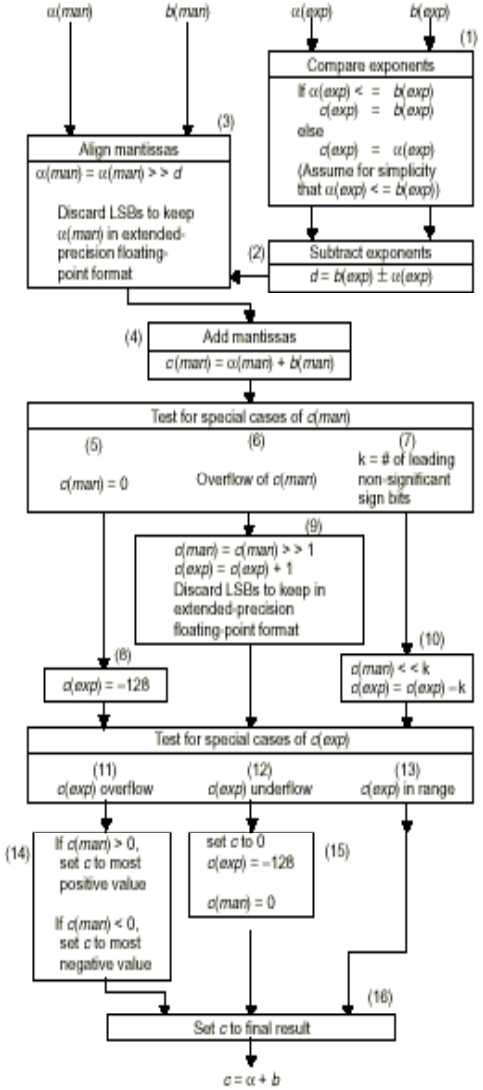
What is fixed-point arithmetic?

- **Floating-point arithmetic**
  - For ex.) 0.9*0.55=0.495, 0.9*5500=4950
    0.9*0.5555555=0.49999995
- **Fixed-point arithmetic**
  - For ex.) 0.9*0.55=0.495 (seems OK)
    0.9*5500=? (overflow)
    0.9*0.5555555=0.499 (quantization)
  - Fixed-point arithmetic
    - Range is limited (scaling needed)
    - Precision limited
    - Less complex hardware

Why fixed-point implementation?

- **Fixed-point arithmetic (or integer arithmetic) requires less chip area, less delay, and less bus width (for memory).**
  - Leads to ½ to 1/10 chip area reduction
  - Leads to x2 speed increase.
- **Many embedded processors do not equip floating-point arithmetic unit**
  - A floating-point arithmetic using library requires many (at least a few 10's) cycles.
  - Leads to x10 speed increase or energy reduction.


a(man)
b(man)
a(exp)
b(exp)
(1)
Multiply mantissas
c(man) = a(man) x b(man)
(50-bit result)
(2)
Add exponents
c(exp) = a(exp) + b(exp)
Test for special cases of c(man)
(3)
c(man) = 0
(4)
Right shift 1
to normalize
(5)
Right shift 2
to normalize
(6)
No shift
to normalize
(7)
c(exp) = -128
(8)
c(man) > > 1
and
c(exp) = c(exp) + 1
(9)
c(man) > > 2
and
c(exp) = c(exp) + 2
Dispose of extra bits
(10)
Put c(man) in extended pre-
cision floating-point format
Test for special cases of c(exp)
(11)
c(exp) overflow
(12)
c(exp) underflow
(13)
c(exp) in range
(14)
If c(man) > 0,
set c(exp) to most
positive value
If c(man) < 0,
set c(exp) to most
negative value
c(exp) = -128
c(man) = 0
(15)
Set c to final result
(16)
c = a x b


a(man)
b(man)
a(exp)
b(exp)
(1)
Compare exponents
If a(exp) < = b(exp)
c(exp) = b(exp)
else
c(exp) = a(exp)
(Assume for simplicity
that a(exp) < = b(exp))
(3)
Align mantissas
a(man) = a(man) > > d
Discard LSBs to keep
a(man) in extended-
precision floating-
point format
(2)
Subtract exponents
d = b(exp) ± a(exp)
(4)
Add mantissas
c(man) = a(man) + b(man)
Test for special cases of c(man)
(5)
c(man) = 0
(6)
Overflow of c(man)
(7)
k = # of leading
non-significant
sign bits
(9)
c(man) = c(man) > > 1
c(exp) = c(exp) + 1
Discard LSBs to keep in
extended-precision
floating-point format
(8)
c(exp) = -128
(10)
c(man) < < k
c(exp) = c(exp) - k
Test for special cases of c(exp)
(11)
c(exp) overflow
(12)
c(exp) underflow
(13)
c(exp) in range
(14)
If c(man) > 0,
set c to most
positive value
If c(man) < 0,
set c to most
negative value
set c to 0
c(exp) = -128
c(man) = 0
(15)
(16)
Set c to final result
c = a + b

Why is fixed-point implementation important in DSP?

- **Digital signal processing**
  - Requires a large number of arithmetic operations (multiply, add, memory access)
    - 10M ~ 100M operations / sec
  - Do not require exact results, in terms of SQNR 40dB~100dB
- **Embedded systems**
  - Many do not equip floating-point arithmetic units
  - Require different word-length according to the applications
    - Audio: around 20 ~ 24 bits
    - Speech: 12~20 bits
    - Video: 8~16 bits
    - Graphics: high precision

Issues in fixed-point optimization

- **Performance estimation**
  - By analytical methods (theory)
    - Usually limited to linear systems
  - By simulation
    - Requires simulation using a large number of input data (fast simulation required)
    - Easy simulation program generation required
- **Automatic scaling**
  - Scales the input and output data (x 2^{-n})
  - 0.9*5500=? -> 0.9*0.55 = 0.495 (*10^{+4})
- **Word-length optimization**
  - Smaller word-length degrades the system performance
  - Larger word-length requires more chip area and power consumption.

Overview of this talk

1. Fixed-point arithmetic and system design
2. Fixed-point simulation method
3. Autoscaler (floating-point to integer)
4. Fixed-point optimizer (wordlength opt.)

Fixed-point Arithmetic and System Design

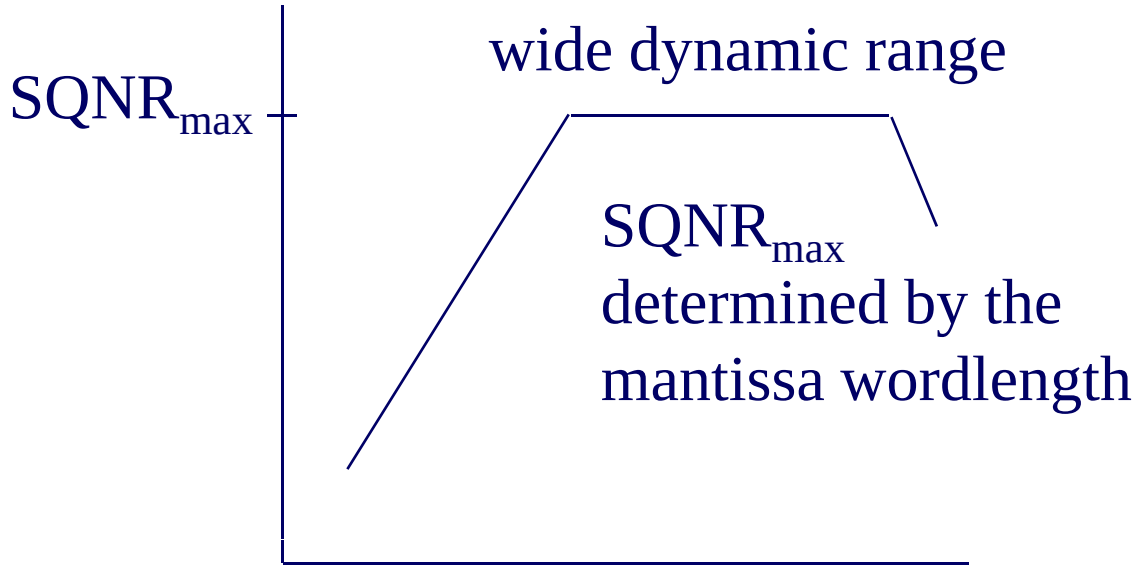
School of Electrical Engineering
Seoul National University

1. Number Representation

- **Floating-point format**
  - $x = M(x)\ 2^{E(x)}$
  - wide dynamic range <- no explicit scaling needed
  - good for algorithm develop, higher hardware cost
- **Fixed-point format**
  - only M(x) is used with constant E(x)
  - minimizing the wordlength is important for economic hardware implementation

Floating-point format

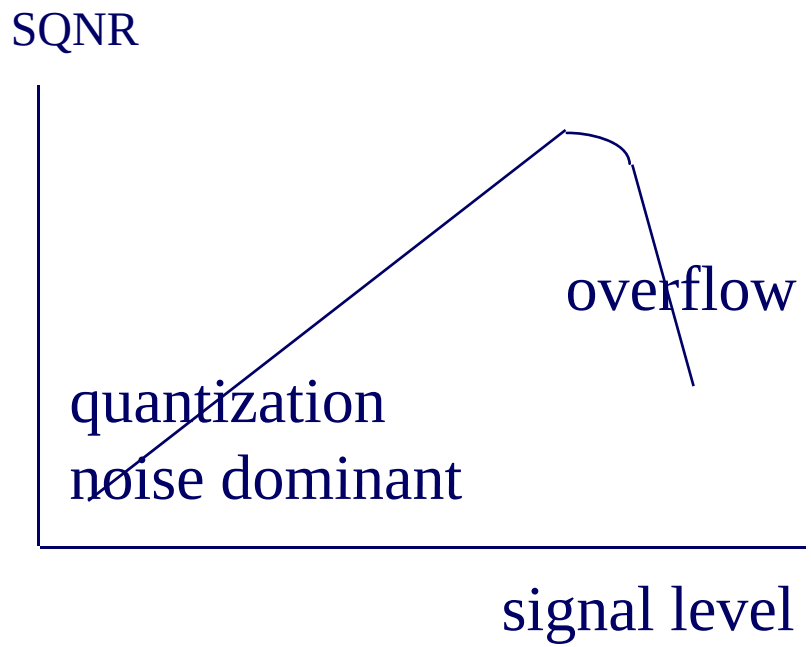
- **SQNR**



- **32bit IEEE standard format is most widely used (24bit mantissa, 8bit exponent)**

Fixed-point format

- **SQNR**
  - Scaling is needed
- **Specification**
  - Negative number representation
  - Overflow handling
  - Word-length reduction
  - Conversion to or from real value



Negative number representation

- **Unsigned**

  0000: 0, 0001:1, 0111:7, 1111:15

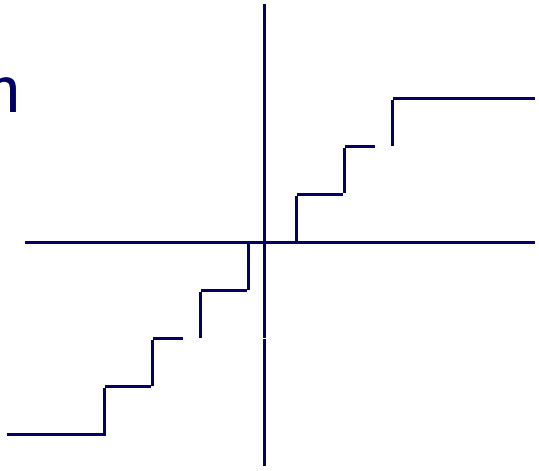
- **Signed**
  - Two's complement ($-x = \overline{x} + 1$)

    0000:0, 0001:1, 0111:7

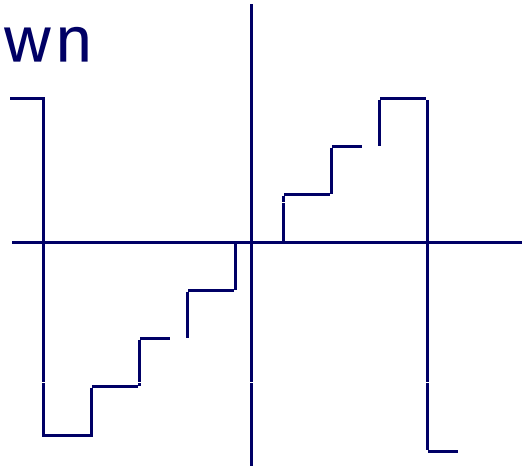
    1111: -1, 1000: -8
  - One's complement ($-x = \overline{x}$)

Overflow handling

- **Saturation**
  - no sign loss for overflow
  - magnitude becomes maximum
  - preferred in signal processing



- **Overflow**
  - simple implementation
  - good when signal range is known

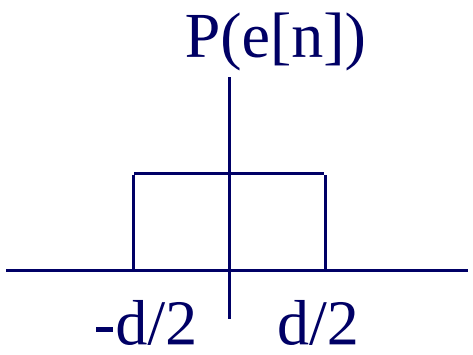


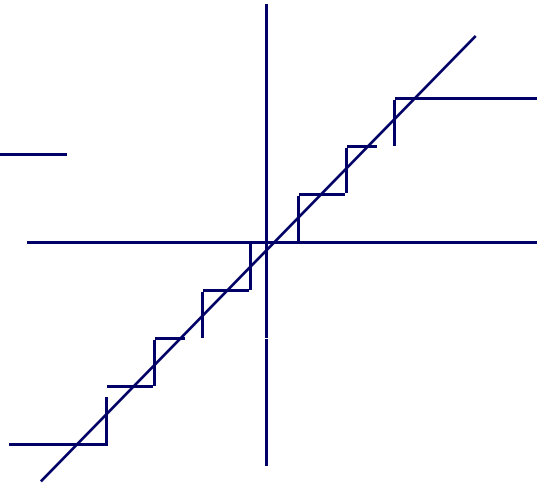
Quantization (Wordlength reduction)

- **Methods**
  - Rounding
  - Truncation
  - Random
  - …
- **Source of Quantization**
  - Signal quantization
    - Generate rounding errors that can be modeled by random noise.
  - Coefficient quantization
    - Deterministic one, affects the frequency response for the case of filters – this is not random noise.

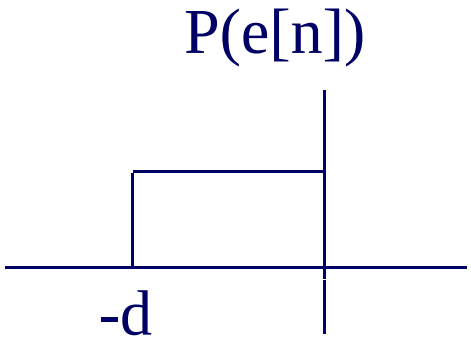
Quantization methods

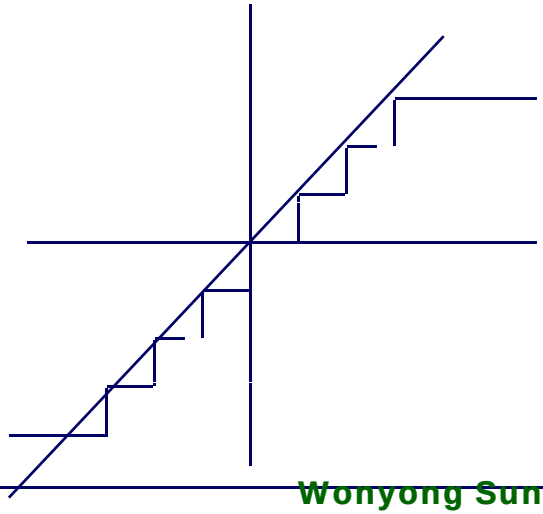
- **Rounding**
  - max. quant. error is d/2





- **Truncation**
  - max. quant. error is d,
  - D.C. bias (fatal when accumulated)



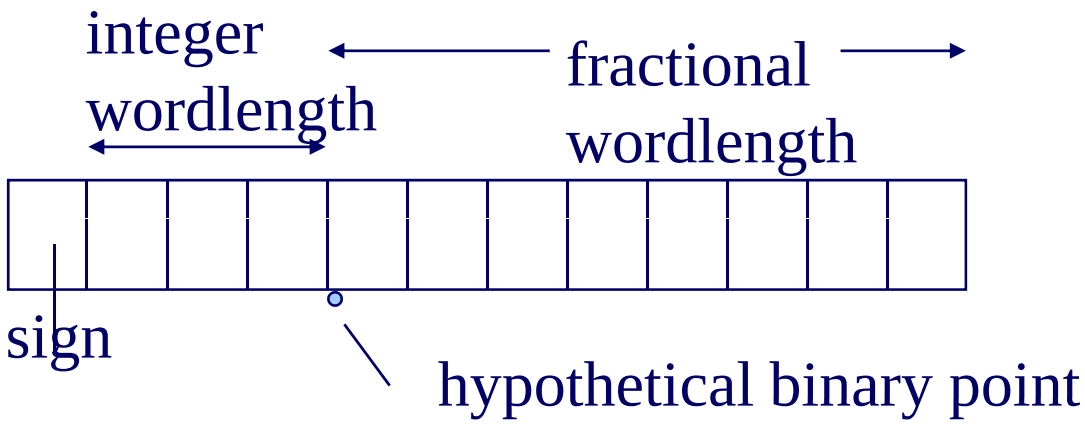


2. Fixed-point Data Format

- **Integer format**
  - all data is interpreted as an integer
  - the quantization level is large (1)
- **Fractional format**
  - all data is between -1 to 1
  - the quantization level is $2^{-(B-1)}$

❖ Generalized fixed-point format

- allow both integer and fractional wordlengths
- integer wordlength determines the maximum signal range, and the fractional wordlength determines the quantization level.



Generalized fixed-point format

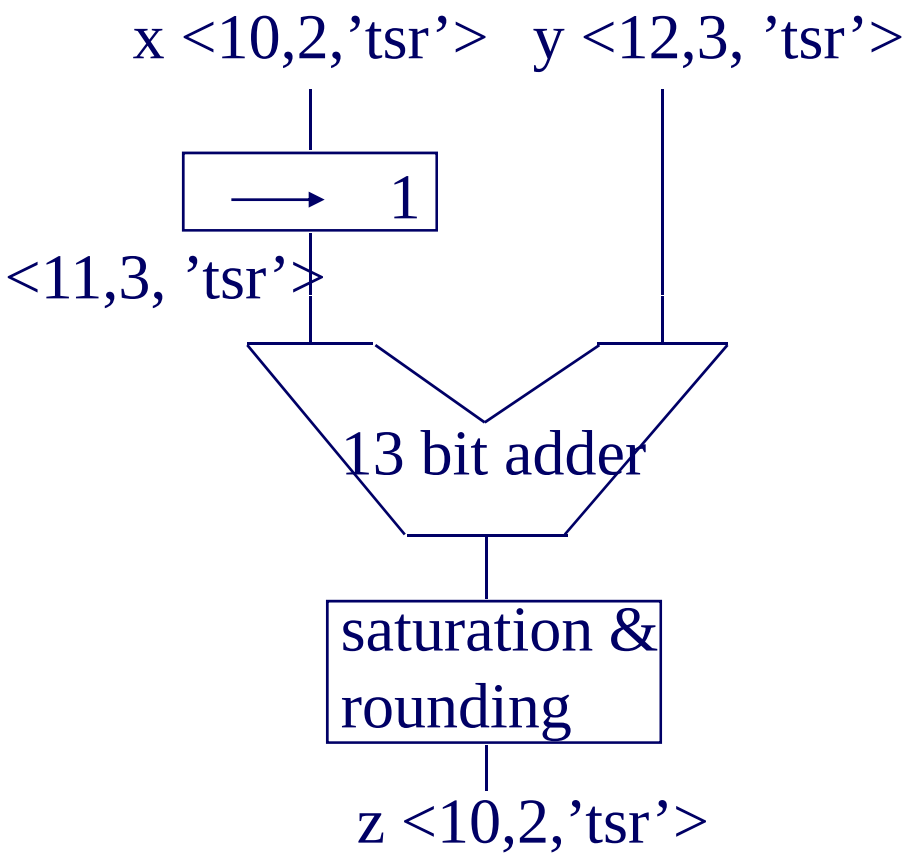
- **SPW format**
  - <wordlength, integer wordlength, sign> <12, 3, t>
  - signal range: $-2^3 \sim +2^3$, quantization level: 2^{-8}
- **Silage (DFL) format**
  - fix <wordlength, fractional wld>; always two's compl.
  - fix <12, 8>
- **SNU Fixed-point Simulator format**
  - <wordlength, integer wordlength, sign_overflow_quantization mode>
  - <12, 3, 'tsr'>

Generalized fixed-point format

- **Arithmetic shift left by n bit**
  - decreases the IWL by n (shift is used for scaling, not for cost effective multiplication)
  - * this is not the case that shift is used instead of multiplication with 2^n
- **Arithmetic shift right by n bit**
  - increases the IWL by n
- **Addition**
  - the IWL of both operands should be the same
- **Multiplication (z = x*y)**

  IWL(z) = IWL(x) + IWL(y) -1; in 2's compl.

Generalized fixed-point format



3. Scaling Method

- **Purpose of Scaling**
  - prevent overflows or waste of bits by proper shifting (arithmetic shift)
- **Scaling implementation**
  - overflow prevention: shift right (IWL increase)
  - save extra-bits: shift left (IWL decrease)
- **Minimum integer wordlength**
  - $IWL_{min}(x) = \lceil \log_2 |R(x)| \rceil$
  - R(x) is the range of a signal

Range estimation methods

- **Analytical method**
  - L1 norm based is most widely used
  - L1 norm based: very conservative estimate
  - applicable to linear or simple systems
- **Simulation based**
  - requires computing power
  - optimum estimate, but input signal dependent
  - applicable to non-linear and time-varying systems

L1 norm based scaling

$$| y[n] | \leq x_{max} \Sigma | h[n] |$$

This means that the output can be higher than the input level by L1 norm times

L1 norm computation

- by using analytical method
- by computing the unit pulse response, and then summing-up the absolute value of the response

L2 norm based scaling

L2 norm = Σ | h [n] |^2

- L2 norm is an upperbound for the signal energy
- it is less pessimistic than the L1 norm
- it is optimum estimation when the input is a white random input

L_{inf} norm based scaling

- $L_{inf} = Max\ |H(jw)|$
  - is an upper bound when the input is sinusoid
  - good when the input signal is very highly correlated
  - least pessimistic norm

Simulation based range estimation

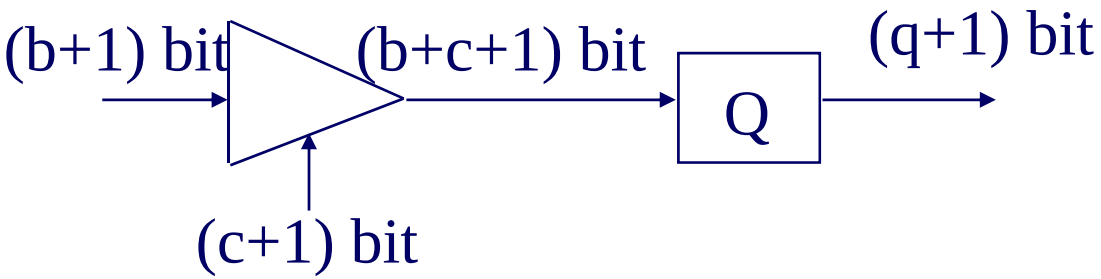
- collect information of sum, squared_sum, absolute_max, and the number of update during the floating-point simulation
- derive mean and standard deviation for a signal after the simulation
- $R(x) = \max\{|m(x)| + n\,\sigma(x), AMax\}$, where n is between 4 to 16

Simulation based range estimation

- **Comparison for a 2nd order IIR filter for speech application (WL = 16 bit)**
  - IWL determined by the simulation based method is 4 bit smaller than that of the L1 norm based method
  - SQNR difference of 24 dB
- **ADPCM implementation**
  - needs 4 different speech files for obtaining a reliable range data

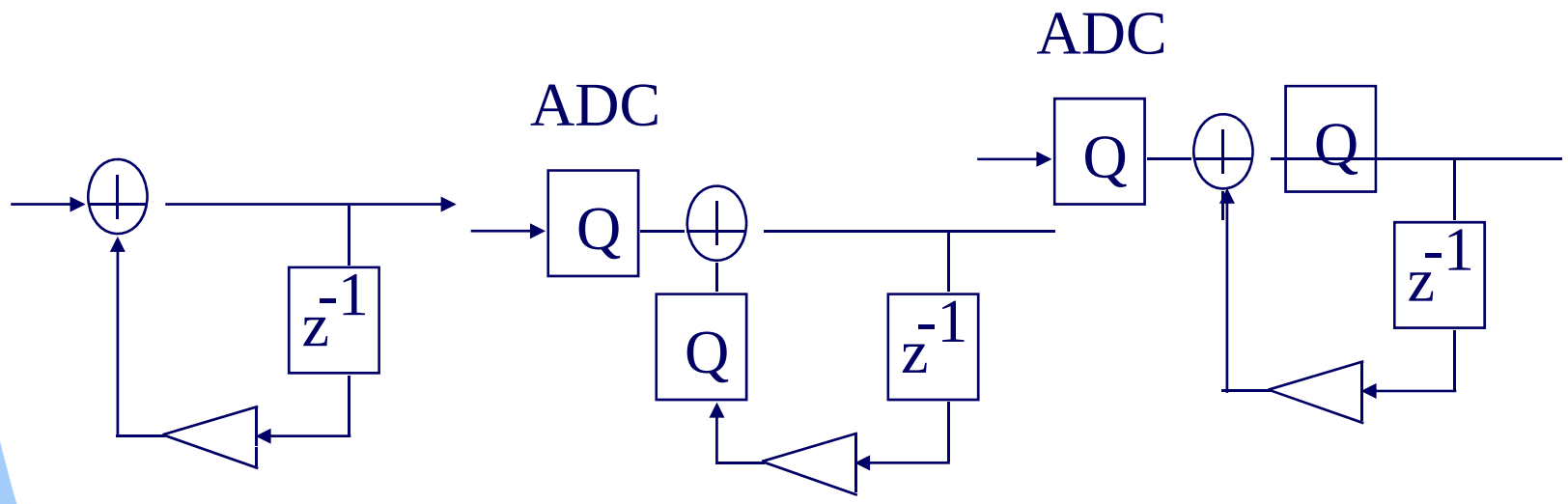
4. Wordlength Determination

- **Wordlength reduction**
  - reduce the wordlength, at an ADC or multiplier output, to minimize the hardware cost while keeping the signal accuracy acceptable

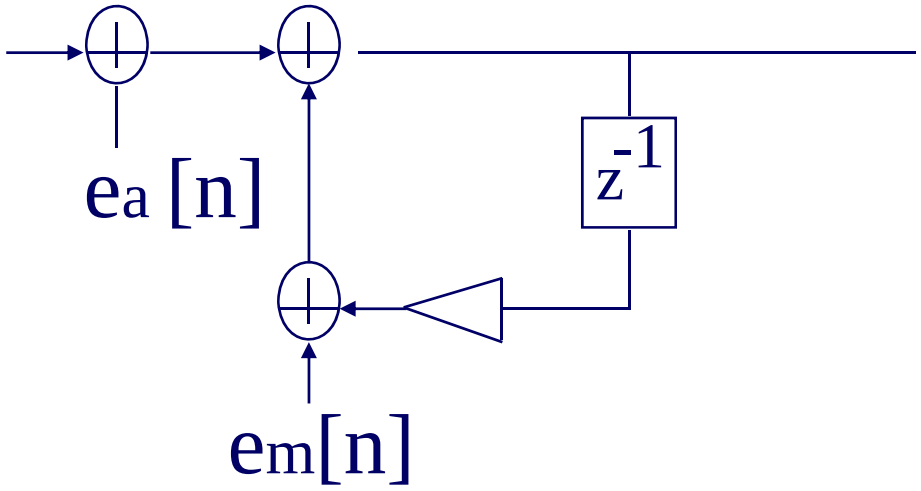


Modeling of quantization

- Ideal system - for floating-point simulation
- Non-linear model - for fixed-point simulation
- Linearized model - for analytical method



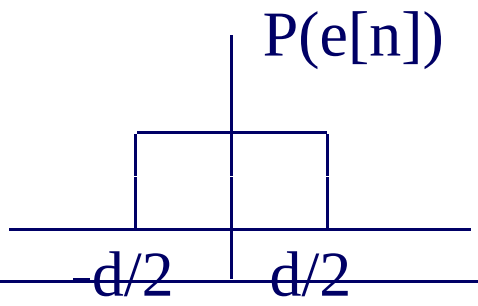
Modeling of quantization - continue



- the magnitude of $e_a[n]$ and $e_m[n]$ is dependent on the fractional word-length

Linearlized modeling of signal quantization

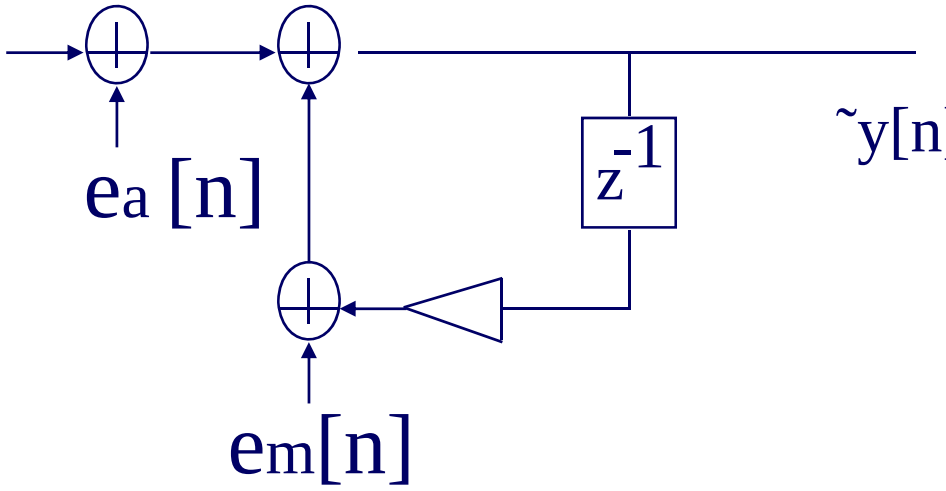
- **Add statistically equivalent quantization error signal instead of the quantizer**
- **Quantization noise**
  - wide-sense stationary white noise
  - uniform distribution between -d/2 to d/2
  - different noise sources are uncorrelated



$$\sigma^2_e = d^2/12$$

Modeling of signal quantization

- **Computation of output noise power**



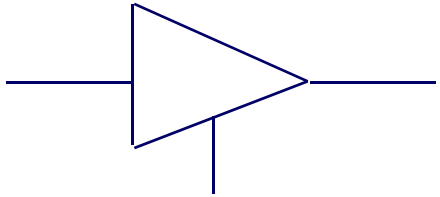
$$\tilde{y}[n] = y[n] + f[n]$$

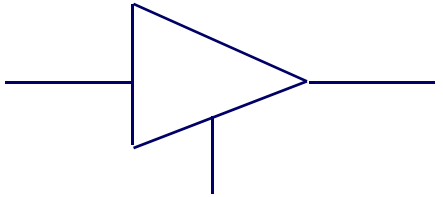
$$\sigma^2_f = (\sigma^2_a + \sigma^2_m)(1/2\pi)\int |H(e^{j\omega})|^2 \, d\omega$$

$$= (\sigma^2_a + \sigma^2_m)\,\Sigma\,|h[n]|^2$$

Coefficient quantization

- **Word-length reduction in filter coefficients or (usually) constant system parameters**





- **Effects of coefficients quantization for FIR and IIR digital filters**
  - deterministic
  - easily known by obtaining the frequency response from the quanitzed coefficients
  - optimization program for minimum hardware cost

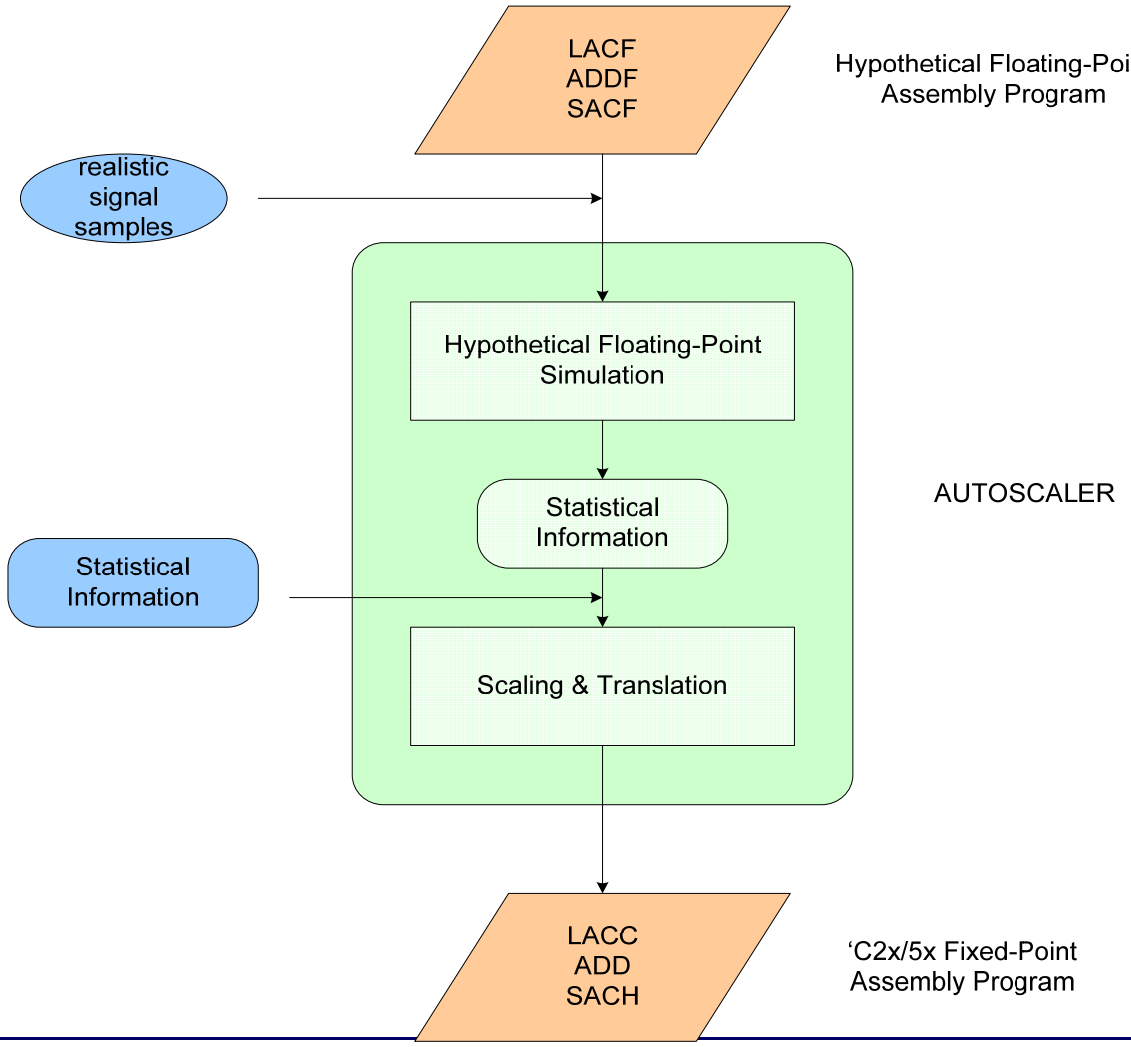
5. Fixed-point Performance

- **Effects of quantization noise in digital signal processing systems**
  - Linear digital filters:
    additive quantization noise at the output
  - Constant filter coefficients:
    transfer function is modified
  - Adaptive digital filter:
    quantization noise changes the adaptation performance or the mean squared error (mse) after the convergence <- distortion

❖Performance measure

- Linear digital filters
  - SQNR
- Adaptive digital filters
  - mean squared error after some time-off period
- Speech coder (waveform coder)
  - the SQNR between the original speech and the reconstructed speech after compensating the filtering or time-delay (but this does not apply to LPC or CELP, model based coder)
- Communication system(Tx-Rx system)
  - the bit-error rate is the best measure, but requires much simulation time
  - the peak error at the eye diagram can be a measure

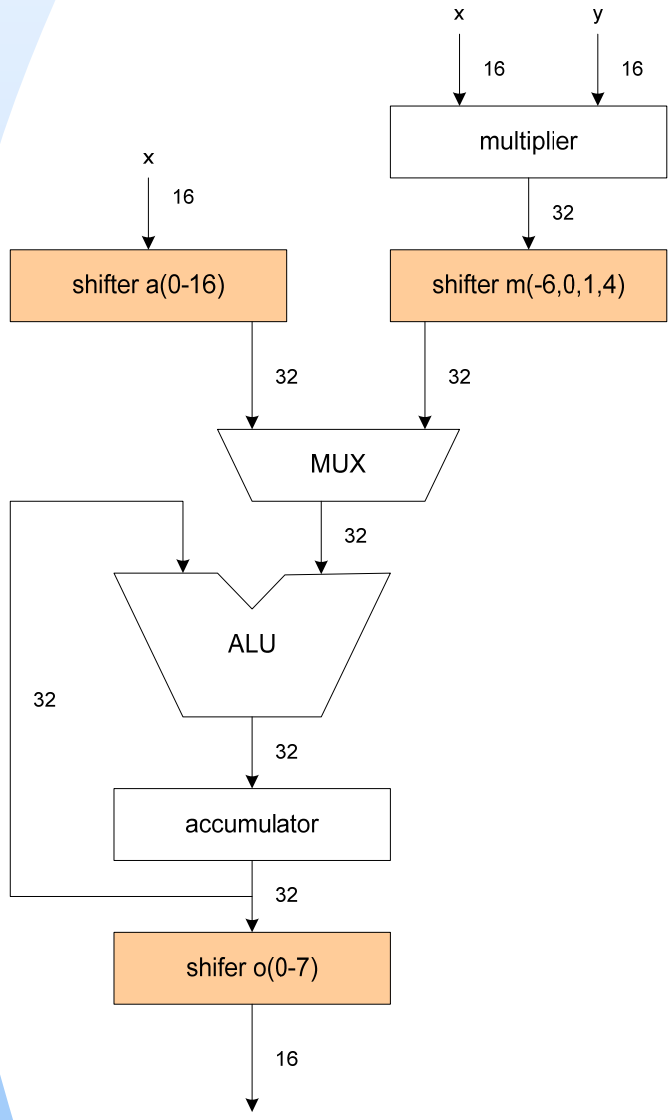
Autoscaler for 'C2x/5x



Hypothetical Floating-Point Instructions

Floating-point instructions		Corresponding fixed-point instructions
Instruction	**Operation**	
ADDF	Add to accumulator	ADD
APACF	Add P register to accumulator	APAC
INF	Input a floating-point data	IN
LACF	Load Accumulator	LAC
LTF	Load T register	LT
LTAF	Load T register, add P reg. to accumulator	LTA
LTDF	Load T register, add P reg. .., move data	LTD
MACF	Multiply and accumulate	MAC
MACDF	Multiply, accumulate, and data move	MACD
MPYF	Multiply	MPY
SACF	Store accumulator	SACH
SACFV	Store with overflow check	macro
SQRTF	Square root	macro
DIVF	Divide	macro

Simplified Data Path of TMS320C2x/5x



- **Note three barrel shifters**
  - Shifter a
  - Shifter o
  - Shifter m

$W_l(t) = \max W_l(x,P,ACC,z)$

$sm = 4, 1, 0, \text{or } -6$

$W_l(ACC) = W_l(w) + W_l(y) + 1 - sm \geq W_l(t)$

$sa[i] = W_l(x[i]) + 16 - W_l(ACC)$

$so = W_l(ACC) - W_l(z)$

Overview of this talk

1. Fixed-point arithmetic and system design
2. Fixed-point simulation method
3. Autoscaler (floating-point to integer)
4. Fixed-point optimizer (wordlength opt.)

Fixed-point Simulation

- Introduction
- Fixed-point format
- C++ fixed-point class based method
- Code conversion method
- Assembly code simulator for C25 DSP

- Reference: Seehyun Kim, Ki-Il Kum, and Wonyong Sung, "Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs," IEEE Tr. Circuits and Systems II,Vol. 45, No. 11, November, 1998.

Introduction

- **Fixed-point simulation is very needed for**
  - Accurate performance estimation
  - But takes much time especially when the wordlength or quantization format is different
- **Seamless conversion from floating-point to fixed-point code is needed**
  - Do not want take much time. Just easy conversion
- **Range estimation code is needed, usually range is obtained from the floating-point simulation**

- **Issues**
  - Word-length determination: high-level design or low-level design?
- **High-level (system or algorithm level) design**
  - Fast simulation and wordlength determination
  - May not be bit-accurate
- **Low-level (assembly, RTL level) design**
  - Slow simulation and word-length determination

2. Fixed-point format

- **Hardware oriented language supports good fixed-point format**
  - VHDL
  - Silage
  - SPW
- **C language does not. Only integer (short, long) and float.**

C++ fixed-point class

- **C++ class – overloading**
- **Each arithmetic operation in C code is translated to the overloaded operations according to the type declaration of the variables or constants**
- **fSig: class for range estimation**
  - Conduct floating-point arithmetic and update the mean, variation, and max of the variable
- **gFix: class for fixed-point arithmetic according to the SNU fixed-point format**
  - gFix a(12, 0, “tsr”)
- **pFix: a pseudo fixed-point library using floating-point ALU: 53bit for mantissa): may not be bit-accurate for high precision data.**

gFix class

```
gFix operator * (const gFix& x, const gFix& y)
// assume that
// result.wl = x.wl + y.wl - 1
// result.iwl = x.iwl + y.iwl
{
   short      iwlen, wlen;
   Integer    I;

   wlen = x.wl + y.wl - 1;
   if (wlen > MAXWL) wlen = MAXWL;
   iwlen = x.iwl + y.iwl;
   I = x.M * y.M;

   return gFix(I,wlen,iwlen);
}
```

3. C++ class based development procedure

- C or C++ programming using floating-point arithmetic -> algorithm verification
- Insert range estimation directives (fSig) to variable declaration statements
- Range estimator -> scaling
- Insert fixed-point simulation directive (gFix or pFix) s to variable declaration statements
- Fixed-point simulator -> wordlength determination

Code example (1st order IIR filter)

```
void
iir1 (short argc, char *argv[])
{
 float Xin;
 float Yout; // fSig()
 float Ydly; // fSig()
 float Coeff;

 Coeff = 0.9;
 Ydly = 0.;
 for (i=0; i< 1000; i++) {
  infile >> Xin;
  Yout = Coeff * Ydly + Xin;
  Ydly = Yout;
  outfile << Yout << '\n';
 }
}
```

(a) Original C++ program

Range estimation & fixed-point simulation code

```
void
iir1 (short argc, char *argv[])
{
 float Xin;
 static fSig Yout("iir1/Yout");
 static fSig Ydly("iir1/Ydly");
 float Coeff;

 Coeff = 0.9;
 Ydly = 0.;
 for (i=0; i< 1000; i++) {
  infile >> Xin;
  Yout = Coeff * Ydly + Xin;
  Ydly = Yout;
  outfile << Yout << '\n';
 }
}
```

(b) Range estimation code

```
void
iir1 (short argc, char *argv[])
{
 gFix Xin(12,0);
 gFix Yout(16,3);
 gFix Ydly(16,3);
 gFix Coeff(10,0);

 Coeff = 0.9;
 Ydly = 0.;
 for (i=0; i< 1000; i++) {
  infile >> Xin;
  Yout = Coeff * Ydly + Xin;
  Ydly = Yout;
  outfile << Yout << '\n';
 }
}
```

(c) Fixed-point simulation code

Comparison of simulation time

- **A Fourth order IIR filter**
  - Floating: 2.19 sec
  - fSig: about twice of the floating-p simulation time
  - gFix: 340 sec
  - pFix: 16.3 sec
  - VHDL: 181 sec
  - SPW: 60 sec

Architecture considerations

- **Fixed-point performance is architecture dependent**

```
gFix a(12, 0, "tsr")
gFix b(12, 0, "tsr")
gFix c(12, 0, "tsr")
gFix d(10, 1, "tsr")

d = a + b + c;
```

```
gFix a(12, 0, "tsr")
gFix b(12, 0, "tsr")
gFix tmp(10, 1, "tsr")
gFix c(12, 0, "tsr")
gFix d(10, 1, "tsr")

tmp = a + b;
d = tmp + c;
```

4. Scaling

- **Can easily be determined from the range information**
  - IWL $\geq$ | R(x)| ceiling

Wordlength determination

- **For uniform wordlength determination**
  - Needs only one iteration of search using fixed-point simulation (16bit –ok-> 12bit –no-> 14bit-ok->13bit…
- **For different wordlength for each variable**
  - It is needed to group the variables that can have the same wordlength
  - Needs to determine the minimum wordlength for each group
  - Then try to find out the combined wordlength satisfying the performance

Overall procedure

- C or C++ programming using floating-point arithmetic
- Integer wordlength determination using fSig class
- Minimum wordlength determination (gFix)
- Optimum wordlength determination (gFix)

Conclusion

- Fixed-point simulation for existing C program can be conducted fairly easily using C++ fixed-point class.
- C program level fixed-point simulation takes much less time than the assembly or RTL level fixed-point simulation, but should be careful for bit-accurate results.

Overview of this talk

1. Fixed-point arithmetic and system design
2. Fixed-point simulation method
3. Autoscaler (floating-point to integer)
4. Fixed-point optimizer (wordlength opt.)

Floating-Point to Integer C Program Converter

School of Electrical Engineering
Seoul National University

Contents

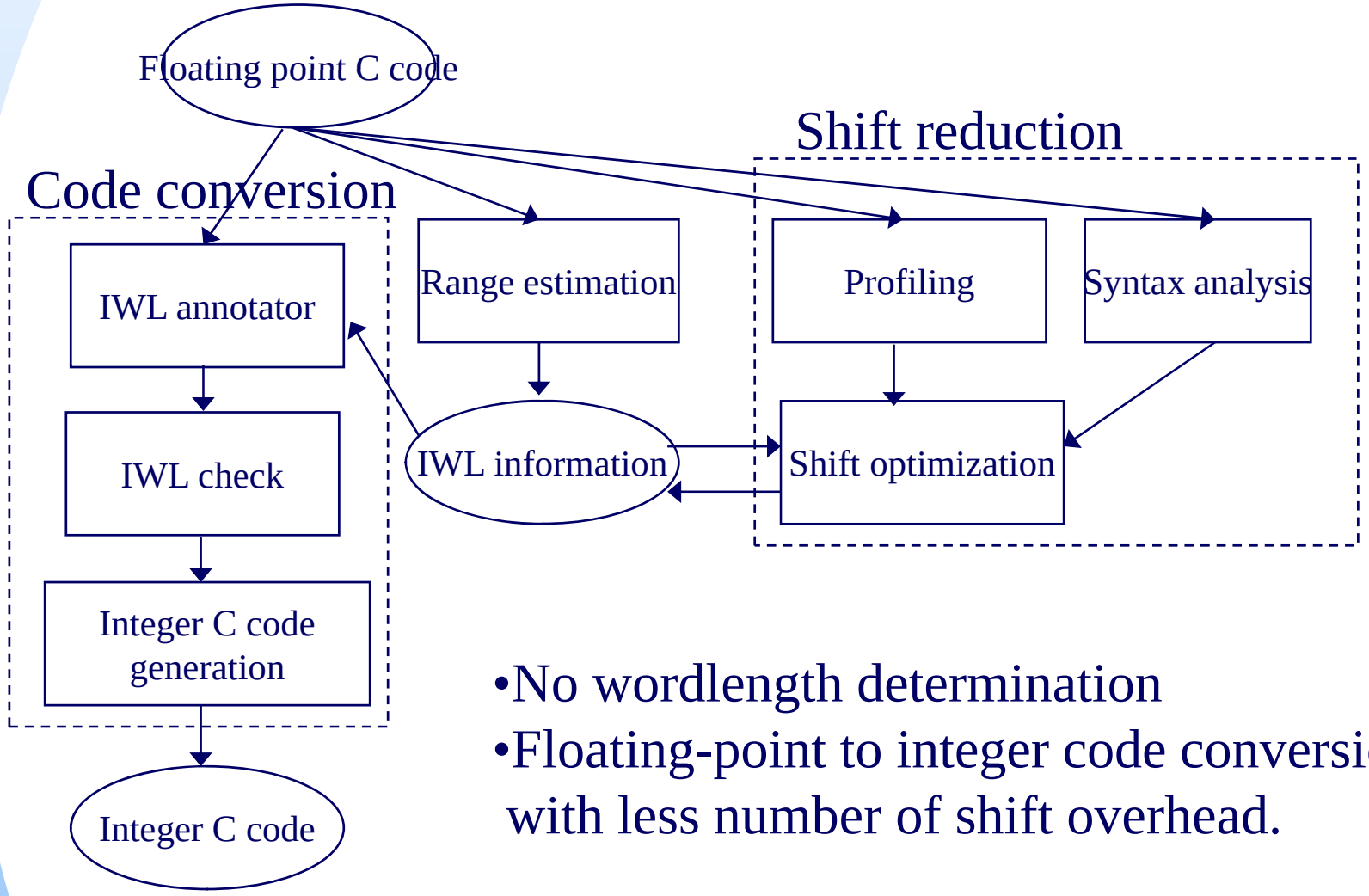
- Introduction
- Simulation based integer word-length determination
- Code conversion
- Shift optimization
- Implementation examples
- Conclusion

- Reference: Ki-Il Kum, Jiyang Kang and Wonyong Sung, "AUTOSCALER for C: An Optimizing Floating-point to Integer C Program Converter for Fixed-point Digital Signal Processors," IEEE Tr. Circuits and Systems II, Vol. 47, No. 9, September 2000, pp. 840-848.

Introduction

- **Motivation**
  - high level language for DSP
    - reduce development time, portability
  - fixed-point DSP
    - scaling is important for avoiding overflows and for accuracy
- **Previous research**
  - autoscaling assembler
  - C++ based fixed-point optimization utility
- **Employed development tool**
  - SUIF compiler system
    - parser, C code generator
    - all processes are performed with the SUIF data structure

Overall design flow



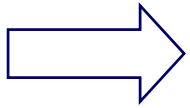
- No wordlength determination
- Floating-point to integer code conversion with less number of shift overhead.

Functional call based fixed-point simulation

- **C++ class based method**
  - applicable to static variables
- **function call based method**
  - applicable to static or automatic variables
    - recursion is allowed
  - automatically generate range estimating code
  - all floating variable in all symbol tables are given its identification number
  - statistic data are collected in a function call inserted for each assignment
  - 2.7 times faster than the C++ class based
  - Implemented using SUIF compiler front end

Original C code

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



Range estimating C code

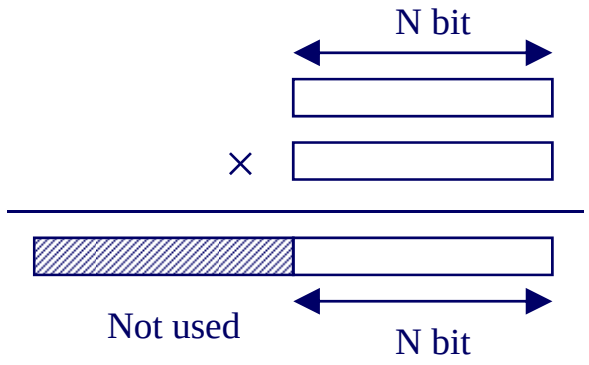
```
main()
{
  init_range(10);
  ...
  c=a+b;
  range(c,3);
  …
}
```

Range estimating library

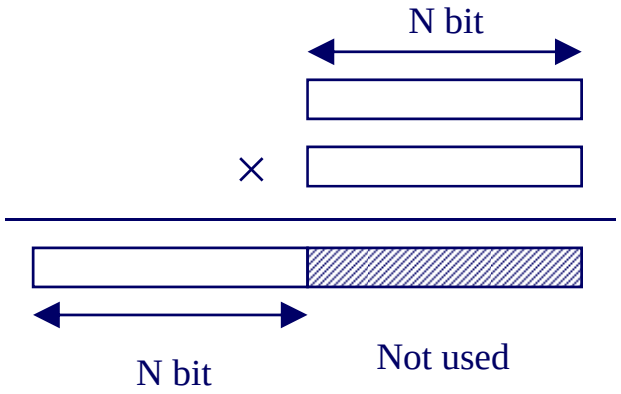
```
range(x,id)
{
  sum[id]+=x;
  sum2[id]+=x*x;
  ...
}
```

Code conversion

- **Float to integer with known integer wordlength**
- **Fixed-point arithmetic rules**
  - add/sub/assign
    - align binary point
  - multiply
    - high word of the multiplied result is used
    - compiler dependent implementation



ANSI C integer multiply



Fixed-point multiply

Fixed-point arithmetic rules.

	Floating -point	fixed-point			result IWL
		Ix>Iy,Iz	Iy>Ix,Iz	Iz>Ix,Iy	
Assignment	x = y	x = y>>(Ix-Iy)	x = y<<(Iy-Ix)	-	Ix
Addition/ Subtraction	z = x+y	x+(y>>(Ix-Iy))	(x>>(Iy-Ix))+y	(x>>(Iz-Ix))+ (y>>(Iz-Iy))	max(Ix,Iy,Iz)
Multiplication	x*y	mulh(x,y)			Ix+Iy+1 or Ix+Iy

z: variable storing the result

Implementation of fixed-point multiplication.

Target processor	Implementation
TMS320C50	#define mulh(x,y) ((x)*(y)>>16)
TMS320C60	#define mulh(x,y) _mpyh(x,y)

Motorola 56000	__inline int mulh(int x, int y) {
	int z;
	__asm("mpy %1,%2,%0":"=D"(z):"R"(x),"R"(y));
	return z;
	}

Code conversion (2)

- **IWL constraints for array and pointer variables**
  - each array element has the same IWL
  - pointer operation with different IWL is prohibited
    - assignment
    - function parameter

Code conversion (3)

- **Implementation using SUIF**
  - IWL annotation
    - read IWL information file and modify symbol table
  - IWL check
    - floating-point variables without IWL info are checked
    - IWL constraints are checked
  - integer C code generation
    - type conversion of variables in symbol tables
    - expression tree conversion by scaling shift insertion

Shift Optimization

- **Scaling shifts are reduced by equalizing the relevant IWL's**
- **Architecture dependent (barrel shifter or not (DSP56000))**
- **Expression conversion**
  - expressions are converted to simple sum of products form

$$x_i = \sum_{j,k} x_j * x_k + \sum_{l} x_l$$

$$x_i = \{\sum_{j,k} ((x_j * x_k) >> s_{j,k}) + \sum_{l} (x_l >> s_l)\} << s_i$$

  - shift amount calculation

$$I_{rhs} = \max_{j,k,l}(I_{x_j} + I_{x_k} + 1, I_{x_l}, I_{x_i})$$

$$s_{j,k} = I_{rhs} - (I_{x_j} + I_{x_k} + 1)$$

$$s_l = I_{rhs} - I_{x_l}$$

$$s_i = I_{rhs} - I_{x_i}$$

Shift optimization (2)

- **Shift overhead is calculated with the IWL of variables**
- **Shift reduction without a barrel shifter**
  - minimize $c_t = \sum_i (d_i + \sum_j e_{i,j}) n_i$
  - constraints

$$x_{k_0} \le x_k \le x_{k_0} + b$$

$$x_k = x_l$$

$$d_i \ge 0$$

$$e_{i,j} \ge 0$$

$$e_{i,j} = d_i + x_k - (x_l + x_m + 1)$$

$$e_{i,j} = d_i + x_k - x_l$$

  - solve using ILP

Shift optimization (3)

- **Shift reduction using a barrel shifter**
  - minimize

$$c_t = \sum_i (f_B(d_i) + \sum_j f_B(e_{i,j}))n_i, \quad f_B(x) = \begin{cases} 1, x \neq 0 \\ 0, x = 0 \end{cases}$$

  - solve using simulated annealing

Shift optimization (4)

- **Implementation using SUIF**
  - profiling
    - insert a function call which collect profiling information after each floating-point expression
  - syntax analysis
    - simplified parse tree is generated
  - shift optimization
    - generate a constraint file for ILP solver
    - generate a simulated annealing program
  - read initial IWL information determined by range estimation, write back optimized IWL information

Implementation examples

- **Fourth order IIR filter**

Floatig-point C code

```
float a1[3] = { 1, 0.355407, 1.0 };
float a2[3] = { 1, -1.091855, 1.0 };
float b1[2] = { 1.66664, -0.75504 };
float b2[2] = { 1.58007, -0.92288 };
float d1[2], d2[2];
void iir4(float *x, float *y)
{
        float x1, y1, t1, t2;
        x1 = 0.01* *x;
        t1 = x1 + b1[0]*d1[0] + b1[1]*d1[1];
        y1 = a1[0]*t1 + a1[1]*d1[0] + a1[2]*d1[1];
        d1[1] = d1[0];
        d1[0] = t1;
        t2 = y1 + b2[0]*d2[0] + b2[1]*d2[1];
        *y = a2[0]*t2 + a2[1]*d2[0] + a2[2]*d2[1];
        d2[1] = d2[0];
        d2[0] = t2;
}
```

Integer C code

```
#define sll(x,y) ((x)<<(y))
int a1[3] =  { 1073741824, 381615360, 1073741824  };
int a2[3] =  { 1073741824, -1172370379, 1073741824  };
int b1[2] =  { 1789541073, -810718027  };
int b2[2] =  { 1696587243, -990934855  };
int d1[2];
int d2[2];
extern void iir4(int *x, int *y)
{
    int x1;
    int y1;
    int t1;
    int t2;
    x1 = sll(mulh(1374389534, *x), 2);
    t1 = sll((x1 >> 4) + mulh(*b1, *d1) + mulh(b1[1], d1[1]), 2);
    y1 = mulh(*a1, t1) + mulh(a1[1], *d1) + mulh(a1[2], d1[1]);
    d1[1] = *d1;
    *d1 = t1;
    t2 = sll((y1 >> 4) + mulh(*b2, *d2) + mulh(b2[1], d2[1]), 2);
    *y = sll(mulh(*a2, t2) + mulh(a2[1], *d2) + mulh(a2[2], d2[1]), 3);
    d2[1] = *d2;
    *d2 = t2;
}
```

Implementation examples (2)

Performance comparison for the fourth order IIR filter.

	# of cycles			SQNR	
	floating-p.	integer	speed-up	floating-p.	integer
’C50	2,980	100	29.8	-	49.3dB
’C60	3,659	9	406.6	-	57.9dB
56000	26,282	921	28.5	-	78.5dB

Implementation examples (2)

The shift reduction results of the fourth order IIR filter.

IWL increment upper bound		0 (no shift reduction)	3	Infinite
# of shifts in C codes		7	4	2
’C50	# of cycles	100	96	94
	speedup	-	4%	6%
	SQNR	49.3dB	51.2dB	54.1dB
’C60	# of cycles	9	6	8
	speedup	-	33%	11%
	SQNR	57.9dB	57.1dB	54.2dB
# of shifts in C codes		5	3	2
56000	# of cycles	921	675	577
	speedup	-	27%	37%
	SQNR	78.5dB	78.5dB	78.5dB

Implementation examples (3)

- **QCELP**
  - 16 source and 4 header files, total of 3648 lines
  - 381 floating-point variables
  - floating-point code: SQNR 17.9 dB
  - converted integer code: 17.36 dB
  - speedup: 24.6 times faster
  - shift optimization
    - shift cost is reduced by 88%
    - 4.5 % speedup
    - 0.1 dB performance degradation

Conclusion

- Read and generate ANSI C compliant programs
- IWL information is kept in separate file.
- Target specific fixed-point multiplication and scaling shift optimization
- Converted integer C codes are 5 - 400 times faster than the floating-point C codes
- Fast simulation due to high level language simulation

$$I_{rhs} = \max_{j,k,l}(I_{x_j} + I_{x_k} + 1, I_{x_l}, I_{x_i})$$

$$s_{j,k} = I_{rhs} - (I_{x_j} + I_{x_k} + 1)$$

$$s_l = I_{rhs} - I_{x_l}$$

$$s_i = I_{rhs} - I_{x_i}$$