Quantum Architecture

EE599-001 & EE699-010, Spring 2026

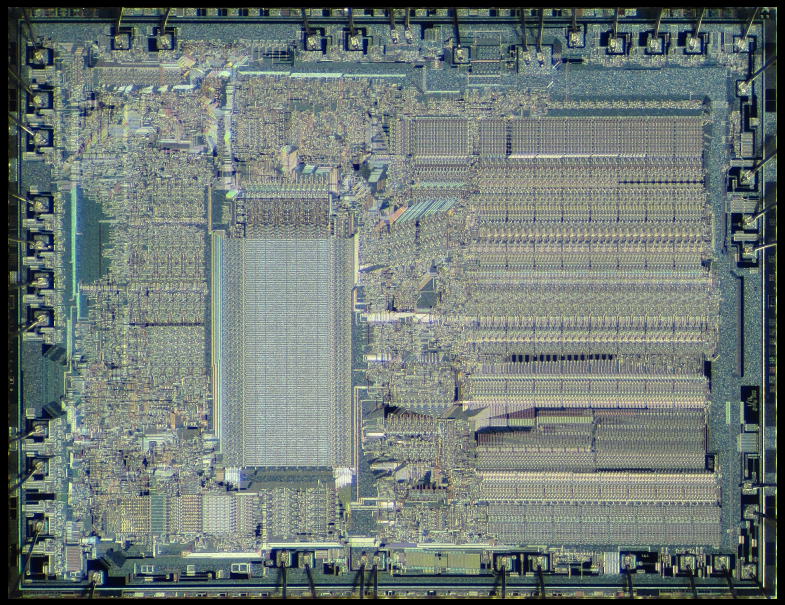
Hank Dietz

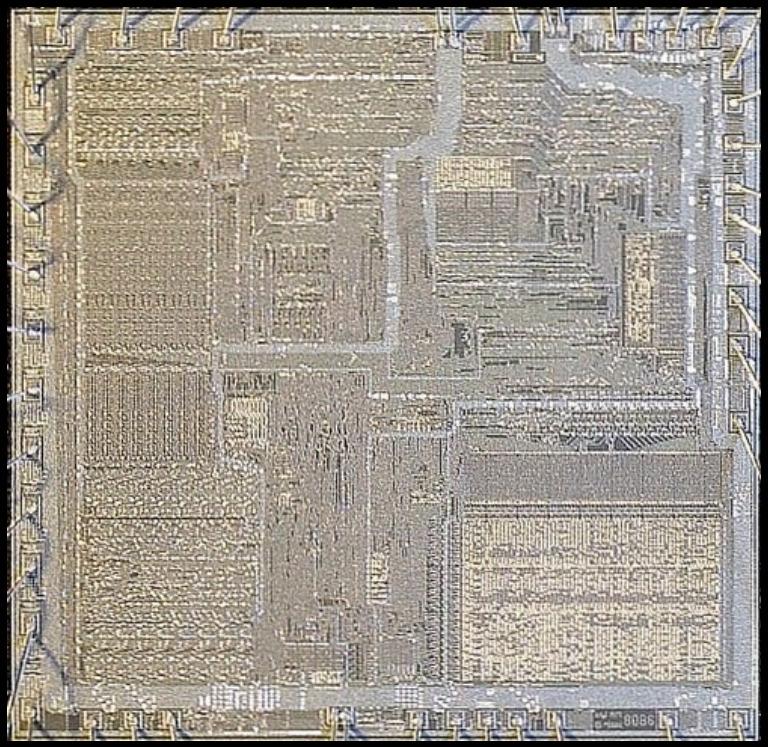
http://aggregate.org/hankd/

University of Kentucky

Attached Processors

Intel's 8087 was hosted by an 8086





Inside A Modern Processor

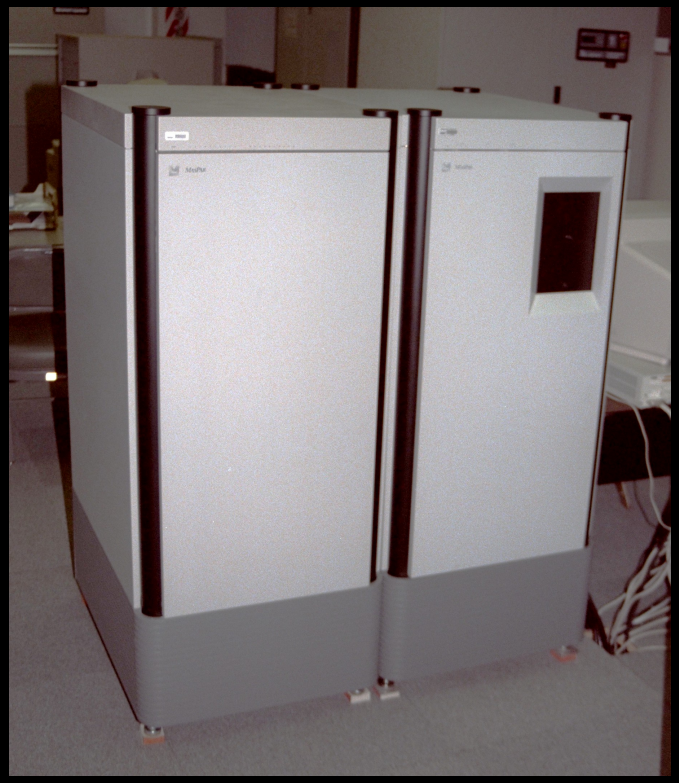
Intel Alder Lake – ME (Management Engine), BSP (BootStrap Processor), ...

10nm ESF/Intel 7 Alder Lake die shot (~209mm²) from Intel via Andreas Schilling on Twitter:
https://twitter.com/aschilling/status/1453391035577495553

Die shot interpretation by Locuza, October 2021

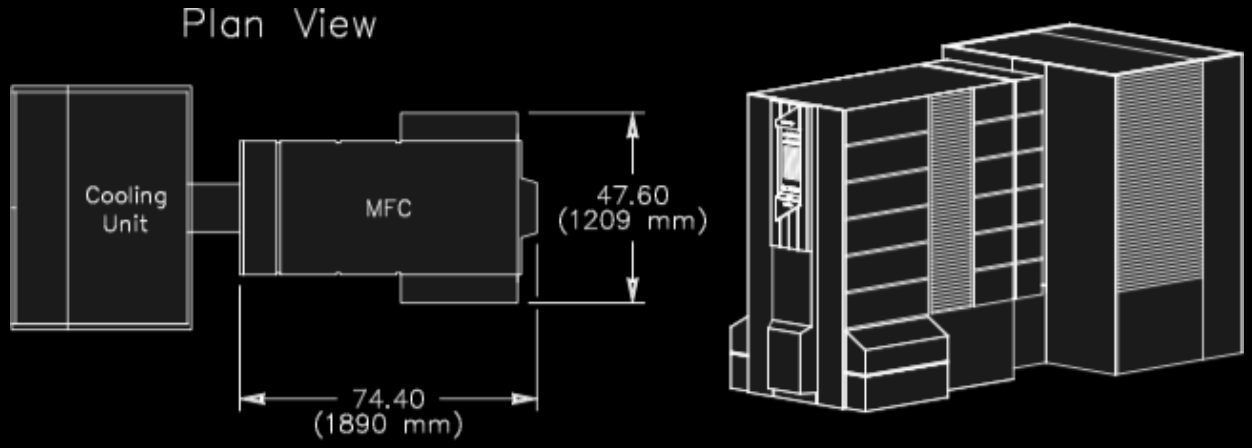
Supercomputers

MasPar MP1
DEC Alpha workstation



Supercomputers

Cray T3D
Apple PowerBook
Y-MP/C90



Supercomputers

Clusters in 108 Marksbury, **Head Nodes**

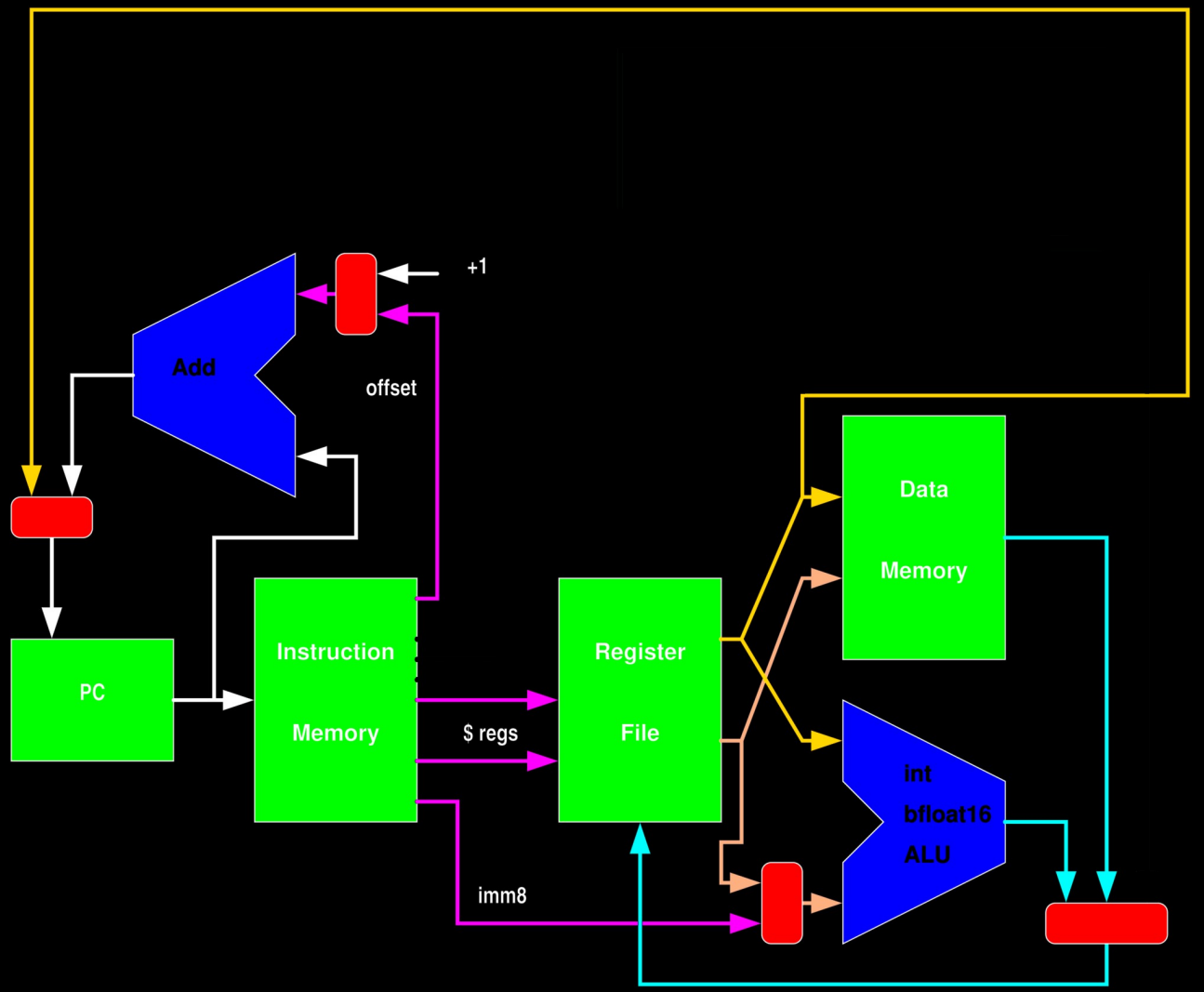
Supercomputers

GPUs (photo from Tom's Hardware)

Supercomputers

- *Supercomputers are like Ogres*
  - There is a parallel computation unit
  - There is a host system with I/O, etc.
  - There is a boot system
  - There is a status display / service processor
- **Quantum Processing Units** (**QPU**s) are just another layer...

Tangled


+1
Add
offset
PC
Instruction
Memory
$ regs
Register
File
Data
Memory
int
bfloat16
ALU
imm8

Tangled Instruction Set

Table 1: Tangled Base Instruction Set

Instruction	Description	Functionality
`add $d,$s`	int add	$d+=$s
`addf $d,$s`	bfloat16 add	$d+=$s
`and $d,$s`	bitwise AND	$d=AND($d,$s)
`brf $c,lab`	branch false to lab	if (!$c) PC+=offset
`brt $c,lab`	branch true to lab	if ($c) PC+=offset
`copy $d,$s`	copy	$d=$s
`float $d`	int to bfloat16	$d=(bfloat16)$d
`int $d`	bfloat16 to int	$d=(int)$d
`jumpr $a`	jump to register	PC=$a
`lex $d,imm8`	load sign extended	$d={{8{imm8[7]}},imm8}
`lhi $d,imm8`	load high	$d[15:8]=imm8
`load $d,$s`	load	$d=memory[$s]
`mul $d,$s`	int multiply	$d*=$s
`mulf $d,$s`	bfloat16 multiply	$d*=$s
`neg $d`	int negate	$d=(-$d)
`negf $d`	bfloat16 negate	$d=(-$d)
`not $d`	bitwise NOT	$d=NOT($d)
`or $d,$s`	bitwise OR	$d=OR($d,$s)
`recip $d`	bfloat16 reciprocal	$d=1.0/$d
`shift $d,$s`	shift left/right	$d=$d<<$s
`slt $d,$s`	set less than	$d=($d<$s)
`store $d,$s`	store	memory[$s]=$d
`sys`	system call	
`xor $d,$s`	bitwise XOR	$d=XOR($d,$s)

Table 2: Tangled Pseudo-Instructions (Macros)

Instruction	Description	Functionality
`br lab`	branch to lab	PC+=offset
`jump lab`	jump to lab	PC=lab
`jumpf $c,lab`	jump false to lab	if (!$c) PC=lab
`jumpt $c,lab`	jump true to lab	if ($c) PC=lab
`load $d,imm16`	load immediate	$d=imm16

Tangled Instruction Set

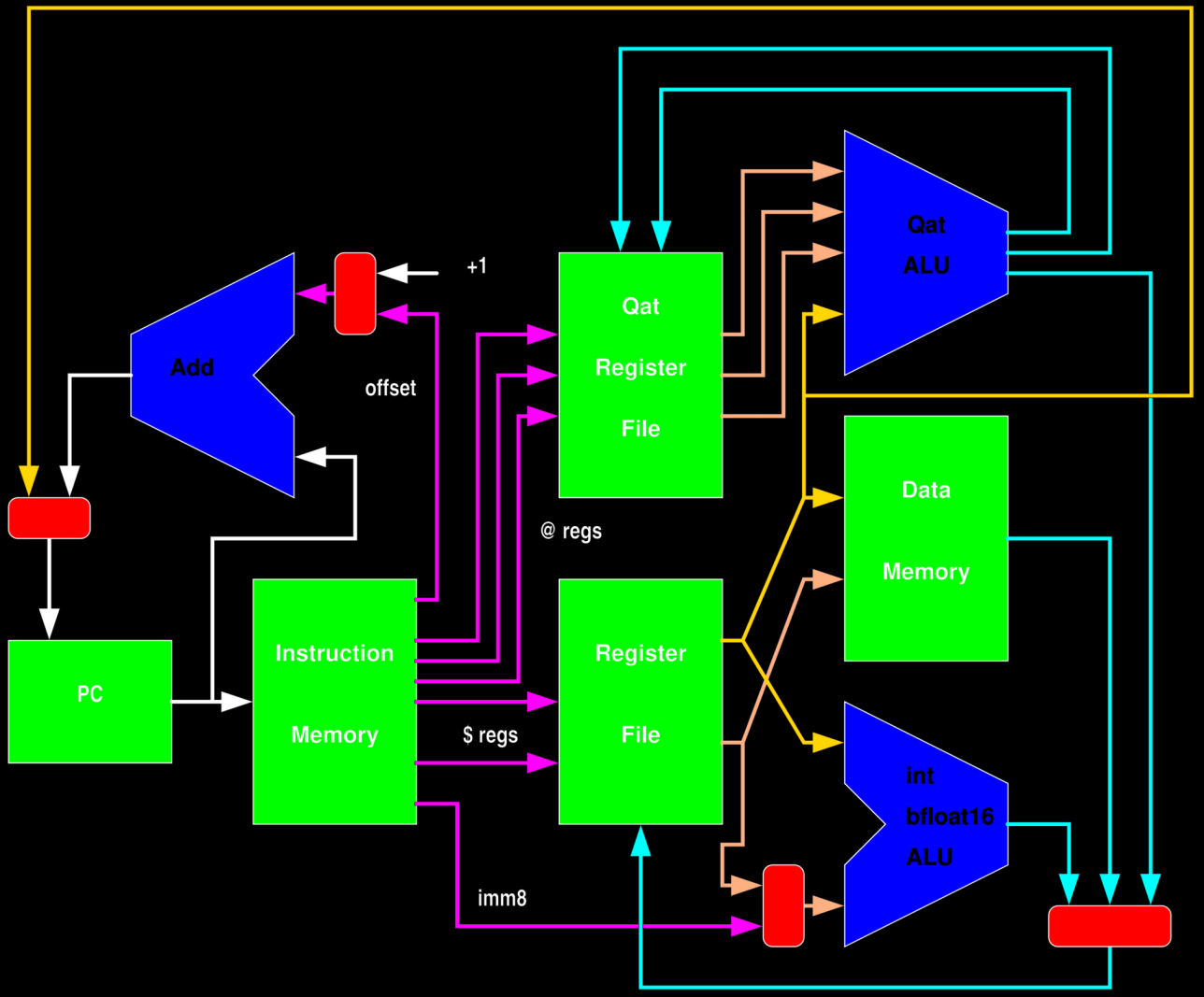
- Ordinary instructions
  - Integer arithmetic and logic
  - Floating-point arithmetic (16-bit bfloat)
  - Control flow
  - Memory access
  - System call

- None of that touches Qat!
  It is a completely normal instruction set...

Tangled Instruction Set

Instruction	Bits	Description	Functionality
`add $d, $s`	16	Add 16-bit integers	`$d[15:0] += $s[15:0]`
`addf $d, $s`	16	Add 16-bit floats	`$d[15:0] += $s[15:0]`
`and $d, $s`	16	bitwise AND 16-bit	`$d[15:0] &= $s[15:0]`
`br addr`	pseudo	BRanch (unconditional), 8-bit offset	`PC = addr`
`brf $c, addr`	16	BRanch False (zero), 8-bit offset	`if ($c[15:0] == 0) PC += (addr-PC)`
`brt $c, addr`	16	BRanch True (non-zero), 8-bit offset	`if ($c[15:0] != 0) PC += (addr-PC)`
`copy $d, $s`	16	COPY	`$d = $s`
`float $d`	16	16-bit integer to FLOAT	`$d[15:0] = (float)$d[15:0]`
`int $d`	16	16-bit float to INTeger	`$d[15:0] = (int)$d[15:0]`
`jump addr`	pseudo	JUMP to 16-bit address	`PC = addr`
`jumpf $d, addr`	pseudo	JUMP False (zero) to 16-bit address	`if ($d == 0) PC = addr`
`jumpr $a`	16	JUMP Register	`PC = $a[15:0]`
`jumpt $d, addr`	pseudo	JUMP True (non-zero) to 16-bit address	`if ($d != 0) PC = addr`
`lex $d, imm8`	16	Load sign EXtended 8-bit to 16-bit	`$d = ((imm8 & 0x80) ? 0xff00 : 0) | (imm8 & 0xff)`
`lhi $d, imm8`	16	Load HIgh 8-bits with 8-bit immediate	`$d[15:8] = imm8`
`load $d, $s`	16	LOAD	`$d[15:0] = memory[$s[15:0]]`
`load $d, imm16`	pseudo	LOAD immediate 16-bit value	`$d = imm16`
`mul $d, $s`	16	Multiply 16-bit integers	`$d[15:0] *= $s[15:0]`
`mulf $d, $s`	16	Multiply 16-bit Float	`$d[15:0] *= $s[15:0]`
`neg $d`	16	Negate 16-bit integer	`$d[15:0] = -$d[15:0]`
`negf $d`	16	Negate 16-bit Float	`$d[15:0] = -$d[15:0]`
`not $d`	16	bitwise NOT 16-bit	`$d[15:0] = ~$d[15:0]`
`or $d, $s`	16	bitwise OR 16-bit	`$d[15:0] |= $s[15:0]`
`recip $d`	16	RECIProcal 16-bit float	`$d[15:0] = 1.0 / $d[15:0]`
`shift $d, $s`	16	shift 16-bit	`$d[15:0] = (($s[15:0] > 0) ? ($d[15:0] << $s[15:0]) : ($d[15:0] >> -$s[15:0]))`
`slt $d, $s`	16	Set Less Than 16-bit integer	`$d[15:0] = (($d[15:0] < $s[15:0]) ? 1 : 0)`
`sltf $d, $s`	16	Set Less Than 16-bit Float	`$d[15:0] = (($d[15:0] < $s[15:0]) ? 1 : 0)`
`store $d, $s`	16	STORE	`memory[$s[15:0]] = $d[15:0]`
`sys`	16	SYStem call to OS	
`xor $d, $s`	16	bitwise XOR 16-bit	`$d[15:0] ^= $s[15:0]`

Tangled & Qat


+1
Add
offset
Qat
Register
File
Qat
ALU
Data
Memory
@ regs
PC
Instruction
Memory
$ regs
Register
File
int
bfloat16
ALU
imm8

Qat Instruction Set

- Quantum-inspired instructions, but:
  - No integer arithmetic
  - No floating-point arithmetic
  - No control flow
  - No memory access
  - No system call

- Only `meas` and `next` touch Tangled state!
  Entangled superpositions live entirely in Qat

NVQLink and CUDA-Q

- NVIDIA's **CUDA** is already a very popular API dealing with GPUs as **Attached Processors**
  - Host vs. Device decomposition
  - Queue system for sending requests to run Device Kernels, DMA copy data, etc.
  - CUDA Kernels are compiled into pseudocode that is translated to native at runtime, analogous to transpiling in QisKit

- Adds hardware interface and extends CUDA

https://www.nvidia.com/en-us/solutions/quantum-computing/nvqlink/

Qat Instruction Set

- Quantum-inspired instructions, but:
  - No integer arithmetic
  - No floating-point arithmetic
  - No control flow
  - No memory access
  - No system call

- Only `meas` and `next` touch Tangled state!
  Entangled superpositions live entirely in Qat

Qat Instruction Set

- Qat operations have their own @ registers
  - Dependences are processed by Tangled
  - No read/write between $ and @ registers
- **Purely combinatorial** with **fixed timing**

Instruction	Bits	Description	Functionality
`and @a, @b, @c`	32	AND	`@a = @b & @c`
`ccnot @a, @b, @c`	32	Controlled-controlled NOT (Toffoli)	`@a ^= (@b & @c)`
`cnot @a, @b`	32	Controlled NOT	`@a ^= @b`
`cswap @a, @b, @c`	32	Controlled swap (Fredkin)	`where (@c) swap(@a, @b)`
`had @a, imm4`	16	HADamard initializer for entanglement	`@a = imm4`
`meas $d, @a`	16	Measure value of entanglement channel	`$d = @a[$d]`
`next $d, @a`	16	Entanglement channel of next 1	`$d = next($d, @a)`
`not @a`	16	NOT (Pauli-X)	`@a = ~@a`
`or @a, @b, @c`	32	OR	`@a = @b | @c`
`one @a`	16	ONE	`@a = 1`
`swap @a, @b`	32	Swap	`swap(@a, @b)`
`xor @a, @b, @c`	32	eXclusive OR	`@a = @b ^ @c`
`zero @a`	16	ZERO	`@a = 0`

Qat Example Program

- Quantum code is basically a straight-line **basic block** executed without pauses

```
pint a = pint_mk(4, 15);  // a=15
pint b = pint_h(4, 0x0f); // b=0..15
pint c = pint_h(4, 0xf0); // c=0..15
pint d = pint_mul(b, c);  // d=b*c
pint e = pint_eq(d, a);   // e=(d==a)
pint f = pint_mul(e, b);  // make non-factors 0
pint_measure(f);          // prints 0, 1, 3, 5, 15
```

Figure 9: Word-level prime factoring of 15.

```
had @0,3
had @1,5
and @2,@0,@1
had @3,4
and @4,@0,@3
had @5,2
and @6,@5,@1
and @7,@4,@6
and @8,@5,@3
had @9,1
and @10,@9,@1
and @11,@8,@10
and @12,@9,@3
had @13,0
and @14,@13,@1
and @15,@12,@14
xor @16,@8,@10
and @17,@15,@16
or  @18,@11,@17
xor @19,@4,@6
and @20,@18,@19
or  @21,@7,@20
and @22,@2,@21
had @23,6
and @24,@0,@23
and @25,@22,@24
xor @26,@2,@21
and @27,@5,@23
and @28,@26,@27
xor @29,@18,@19
and @30,@9,@23
and @31,@29,@30
xor @32,@15,@16
and @33,@13,@23
and @34,@32,@33
xor @35,@29,@30
and @36,@34,@35
or  @37,@31,@36
xor @38,@26,@27
and @39,@37,@38
or  @40,@28,@39
xor @41,@22,@24
and @42,@40,@41
or  @43,@25,@42
had @44,7
and @45,@0,@44
and @46,@43,@45
xor @47,@40,@41
and @48,@5,@44
and @49,@47,@48
xor @50,@37,@38
and @51,@9,@44
and @52,@50,@51
xor @53,@34,@35
and @54,@13,@44
and @55,@53,@54
xor @56,@50,@51
and @57,@55,@56
or  @58,@52,@57
xor @59,@47,@48
and  @60,@58,@59
or   @61,@49,@60
xor  @62,@43,@45
and  @63,@61,@62
or   @64,@46,@63
xor  @65,@61,@62
xor  @66,@58,@59
xor  @67,@55,@56
xor  @68,@53,@54
xor  @69,@32,@33
and  @70,@13,@3
xor  @71,@12,@14
and  @72,@70,@71
and  @73,@69,@72
and  @74,@68,@73
or   @75,@74,@74
not  @75
or   @76,@67,@75
or   @77,@66,@76
or   @78,@65,@77
or   @79,@64,@78
or   @80,@79,@79
not  @80
lex  $0,31
next $0,@80
copy $1,$0
next $1,@80
lex  $2,15
and  $0,$2 ;5
and  $1,$2 ;3
```

Figure 10: Code prime factoring 15 (3 columns).

Quantum Computation

- Starts with a conventional host
- Enables a self-contained combinatorial QPU
  - Entangled superpositions live here
  - Each block executes without interruption (delays could collapse superposition)
- Host decodes instructions and controls QPU (could overlap conventional execution)
- Host (and above) does all I/O conventionally

NVQLink and CUDA-Q

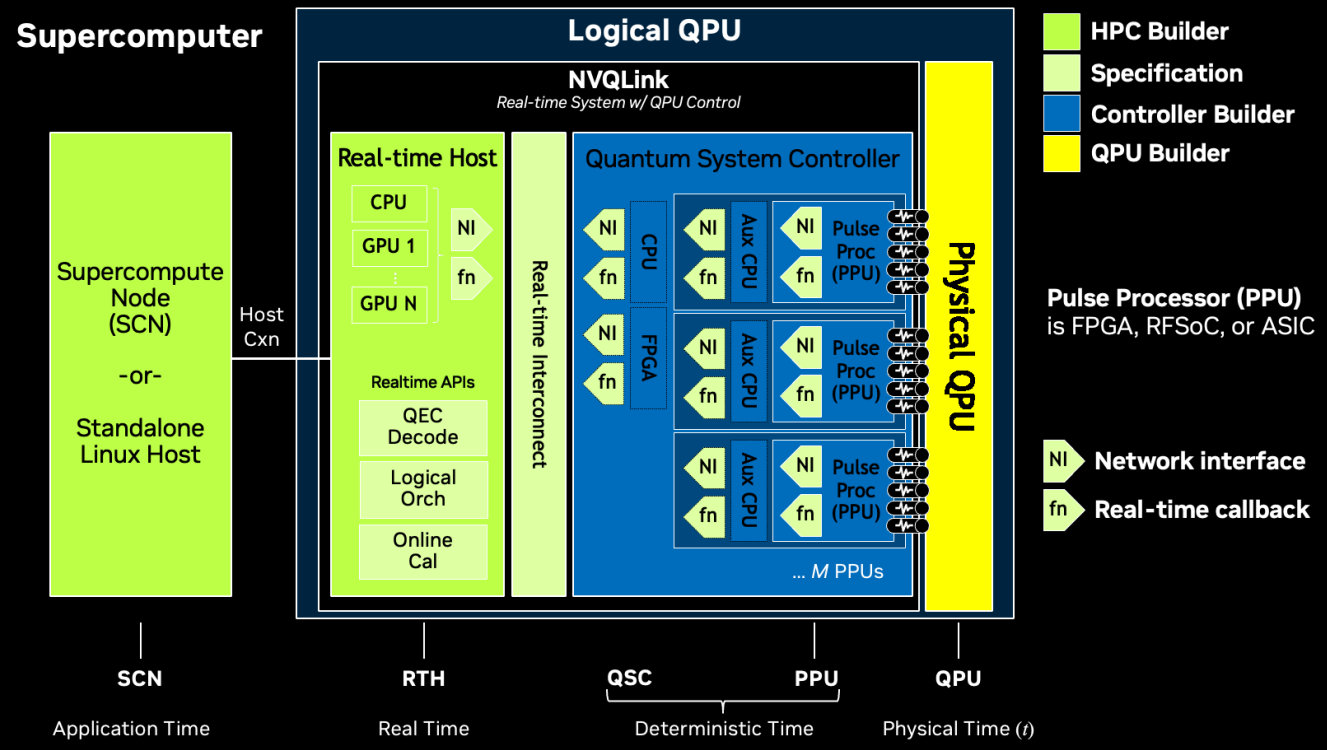
- NVIDIA's **CUDA** is already a very popular API dealing with GPUs as **Attached Processors**
  - **Host** vs. **Device** decomposition
  - Queue system for sending requests to run **Device Kernels**, DMA copy data, etc.
  - CUDA Kernels are compiled into pseudocode that is translated to native at runtime, analogous to transpiling in QisKit

- Adds hardware interface and extends CUDA

https://www.nvidia.com/en-us/solutions/quantum-computing/nvqlink/

NVQLink and CUDA-Q

NVQLink System Architecture



- **NVQLink** is a real-time-control network for GPU to **Quantum System Controller** (**QSC**)

https://developer.nvidia.com/blog/nvidia-nvqlink-architecture-integrates-accelerated-computing-with-quantum-processors/

CUDA-Q

- **CUDA-Q** is a Python/C++ dialect of CUDA that has constructors for quantum circuit kernels
  - Not really very different from QisKit…
  - May handle syndrome error correction using GPU-optimized parallel processing
  - Libraries for various hybrid algorithms
- Many hardware backends
- Reference:

  `https://nvidia.github.io/cuda-quantum/0.14.0/index.html`