Implementation and Design trade-offs

Q. Given a specification how do you find the perfect implementation?
A. There is no such thing!

- There are numerous ways of making something which works.
- (There are many more ways of making something which doesn’t work!)

Consider the implementation (largely) in a top-down fashion

- Algorithms
- Architecture
- RTL
- Practicalities
  - power supply
  - cooling
  - pad limitations
  - …

The ‘right’ algorithm

In this module we are assuming that the algorithm and the higher levels of the architectural design are fixed and we concentrate on the implementation. This section covers the microarchitecture and the RTL; circuit details are left for later.

It is very important to choose an efficient algorithm. For example, to choose what is likely to be a software exemplum, searching 1000 records in a linear table may require an average of 500 comparisons whereas in a binary tree they may only average 9 (slightly more complex) tests.

However clever the later implementation it is unlikely to compensate for such a crushing disadvantage.

Top-down design?

So the principle is that top-down design is usually the best way to go.

However, it is important that when it comes to implement the algorithm this is possible at reasonable cost. Plotting a circle may look simple using sines and cosines but calculating these ‘on the fly’ may be prohibitive. Therefore it is a good plan to look ahead and consider the likely implications before finally committing to an implementation plan!

Example: Bresenham’s line algorithm

Perhaps the ‘obvious’ way to draw a line is to use an equation such as:

\[ y = mx + c \]

and then, by iterating over x work out each y position then round it onto the nearest pixel. This requires the determination of the (fractional) m and c (by division) and a multiplication for each pixel plot. Because ‘real’ numbers are only approximated the result is subject to error.

Bresenham’s algorithm plots the closest possible pixels to a specified line without using (expensive) multiplication or division. It does not require floating point representation and it does not suffer from rounding errors.

Example: The Fast Fourier Transform

First, if you don’t follow the technical details on this page, don’t worry. It is simply an example of how algorithmic level can be the most important place to perform optimization.

The Fast Fourier Transform (FFT) – in computer terms usually attributed to Cooley and Tukey (1965) but the principle stretches back much further – is a means of optimizing a Discrete Fourier Transform (DFT) by combining redundant operations. The Fourier transform is a means of determining the spectral components of a signal – akin to finding the volumes of the notes which make up a chord.

A discrete Fourier transform represents an input by a set of samples and produces an equal number of results. Better resolution is achieved by processing more samples. In its simple form a DFT can be represented by a matrix multiplication.

Clearly this can involve a lot of multiplication. The complexity is \( N^2 \), where \( N \) is the number of samples.

The FFT exploits redundancy in the coefficient matrix to reduce the number of calculations. The complexity of a comparable FFT is \( N \log_2(N) \).

The table, below, shows the sort of savings which can be made. Where such things are possible they can vastly outweigh any optimizations in later implementation.

<table>
<thead>
<tr>
<th>( N )</th>
<th>( N^2 )</th>
<th>( N \log_2(N) )</th>
<th>Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>64</td>
<td>24</td>
<td>62%</td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>160</td>
<td>84%</td>
</tr>
<tr>
<td>256</td>
<td>65536</td>
<td>2048</td>
<td>97%</td>
</tr>
<tr>
<td>1024</td>
<td>1048576</td>
<td>10240</td>
<td>99%</td>
</tr>
</tbody>
</table>

The point is that the algorithm is the first, and often the best, place to optimize.
Design trade-offs

- There are many ways of producing a correct design. (There are even more ways of producing an incorrect one.)
- The ‘best’ design for your application will be influenced by numerous different factors:
  - Speed
  - Power
  - Size
  - Simplicity

- Most of these rely on skill and judgement in the first instance.
  - Later it is possible to get better estimates from tools.
- RTL allows immediate feedback on cycle counts.
  - Experience (or experiment) suggests what may be feasible in a cycle.

"On time, on budget, working; choose any two."

Design trade-offs

Speed

In one respect, ‘the faster, the better’ is a reasonable design philosophy; after all it is easy to reduce the clock rate to slow an implementation down, but the maximum frequency cannot be exceeded.

However there are often ‘real time’ constraints which need to be met, but not exceeded. A real-time circuit needs to guarantee a minimum worst-case performance. For example, there is unlikely to be a benefit in an MP3 decoder being able to go faster than is needed to play a particular track.

Increasing speed beyond a certain level is likely to require a larger, more complex, power-hungry circuit.

Power

Reducing power consumption is important, particularly in battery-powered equipment. Power consumption is (loosely) related both to the size of the device and its speed. Generally, more logic means higher power consumption as does going faster. Note, however, the difference between power and energy.

- Energy is the limited resource stored in the battery.
- Power is how fast energy is ‘used’ over time.

Thus, a circuit which computes at double speed but halts for half the time will use (roughly) the same energy as one which consumes half its power all the time. (There can be some more subtle trade-offs made in this area.)

Remember that the power which is dissipated emerges as heat and this has got to be extracted. Low power therefore can mean better even for ‘tethered’ equipment, by alleviating the need for fans, allowing boxes to be sealed, running cooler (extending product lifetime) and, of course, reducing the electricity bill.

Size

Depending on the application, it may be better to produce a small chip. Chips are manufactured on silicon wafers and the cost/wafer is roughly fixed. A naive approach would suggest that a chip of half the area would therefore cost half as much.

However, because there is only a certain yield of functional chips which is governed by the density of defects on the wafer, a functional chip of 10 mm$^2$ will cost less than half one of 20 mm$^2$. For a device that is to be mass marketed, size can be critical.

When implementing on an FPGA what matters primarily is that the design will fit. However FPGAs come in different sizes (and prices) so a small size usually means a cheaper product.

Even then an FPGA will come with certain resources (such as RAM or multipliers) on board. If these can be exploited there may be savings on flip-flops or logic.

Simplicity

A simple design is likely to be small, which is a benefit in itself. However design simplicity brings other benefits. Primarily, a simple design is easier to build — and much easier to test and debug — which reduced development time and cost. Reduced development time means the product is marketed sooner which brings disproportionate gains in sales. Product reliability helps gain future sales — or, rather, unreliability loses them.

The enemy of simplicity is the need for more (or better) features. For example HDTV processes more pixels per second than older standards so the decoder has to work harder, perhaps demanding a faster or more parallel architecture.
Code style

Example: decade counter
Objective: enable next clock edge when state reaches ‘9’

Desired behaviour:

Alternatives:

```verilog
always @ (posedge clk)
  if (count < 9) count <= count + 1;
  else count <= 0;
assign rc = (count == 9);
```
or

```verilog
always @ (posedge clk)
  if (count < 9) count <= count + 1;
  else count <= 0;
always @ (posedge clk) rc <= (count == 8);
```

NOTE!
The second example will generate an output (slightly) sooner and without glitching:

- faster and possibly safer: may also save power

FSM design

Finite State Machines

Mealy machine

- Outputs may depend on inputs
- Smaller
- Simpler?
- Slower

Moore machine

- Outputs part of state
- Larger
- Faster

Recommendation

The latter style in most cases. By registering the outputs you have a good timing reference to work from – useful when considering the design freedom of subsequent blocks. The immunity from output glitches may be a bonus.

But …

- There is an added cost in logic
- When specifying the output states you need to think ahead!

“From here [current state] I want to go there [next state] and when I’m there I’ll also want these outputs.”

Thus to indicate ‘I am in state 9’ via a flip-flop this has to be detected in state 8. This must also be subject to any other controls (such as an enable).

Coding style

Some people prefer to separate logic and registers

Instead of:

```verilog
always @ (posedge clk)
  if (count < 9) count <= count + 1;
  else count <= 0;
always @ (posedge clk) rc <= (count == 8);
```

this:

```verilog
always @ (count) // The ‘cloud’
begin
  if (count < 9) next_count = count + 1;
  else next_count = 0;
  next_rc = (next_count == 9);
end

always @ (posedge clk) // The ‘rectangle’
begin
  count <= next_count;
  rc  <= next_rc;
end
```

‘carry’ derived from register input.

- More verbose
- Clearer?

You decide what suits you better.
Think sync!

There are very good reasons for keeping to a synchronous style when constructing an electronic design:

- It simplifies the design task
  - Time is quantised
  - Given a state and inputs the next state can be determined unambiguously
  - Logical hazards (glitches during switching) are irrelevant (in general)
- Current toolchains are optimised for synchronous circuits
  - It’s easy to specify synchronous circuits in HDLs
  - Timing analysis and optimization is designed to fit into clock periods
- Implementation technology is designed with synchronous circuits in mind
  - ASIC libraries primarily support D-type flip-flops
  - FPGAs contain many built-in D-type flip-flops

Synchronous design

By this stage you should be used to the concept (and advantages) of synchronous design. The most important aspect is that the logic functions can be designed statically; glitches as logic switches don’t matter as they can’t pass the registers. A clock happens only when everything has settled and everything changes ‘at once’.

A HDL (like Verilog) makes this easy; simply ensure that all state changes use the same clock (and, preferably, the same edge).

Example: sample a counter every second cycle.

**Bad idea:**
```verilog
always @ (posedge clk) clk2 <= !clk2;
always @ (posedge clk) a <= a + 1;
always @ (posedge clk2) b <= a;
```

The timing relationship between the clocks is poorly defined.

Behaviour depends on a race between a and clk2.

**Good idea:**
```verilog
always @ (posedge clk) En <= !En;
always @ (posedge clk) a <= a + 1;
always @ (posedge clk) b <= a;
```

Changes happen simultaneously. The ‘En’ signal controls whether a change will happen or not.

Toolchains are geared towards this style of design so they are capable of ensuring that the flip-flops change (near enough to) simultaneously.

Asynchronous design

It is possible to produce sequential circuits without clocks; indeed much work has been done on this in this School! There are some potential advantages to asynchronous designs.

- No ‘timing closure’ worries – except real-time applications
- Inherent ‘clock gating’ saving power when there is no work to do
- Self-adaptive to conditions, e.g. for Dynamic Voltage Scaling (DVS)
- Spread-spectrum switching reduces emitted electromagnetic noise.

However the disadvantages largely outweigh these.

(If you didn’t follow any of those points, don’t worry about them.)

An asynchronous FSM can be constructed from RS flip-flops. (An RS flip-flop is a simple asynchronous FSM!) An input change may then cause one or more flip-flops to change state, which may cause others to change state, and so on until the system restabilises. Care must be taken to ensure no signal can glitch and cause an erroneous state to occur whilst the logic is in transition.

There are some parts of a system where asynchronous signals are unavoidable. External inputs fall into this category. However the usual strategy is to synchronise these to the internal clock as soon as possible.

As the scope of chips gets larger it becomes convenient to restrict synchronous operation to particular assemblies and worry less about the differences in clocks between them. However the each assembly can be synchronous internally and this is usually a good plan. A little more on this topic is covered under ‘interconnection’.

Counter Register

a  b

clk  clk2

✕

Counter Register

a  b

clk  clk2

✓
Think parallel!

- (Imperative) software programming is essentially serial
- Hardware programming is implicitly parallel
- Increasing parallelism typically:
  - Increases speed
  - Increases resources required
  - Can decrease complexity
  - May decrease power

Techniques may include:

- Subdividing the logic by pipelining
  - Pipelines need to be reasonably balanced
- Pre-evaluating some of the logic in an earlier stage if possible
  - May make the structure messy
- Finding a more parallel logic expression
  - A classic example is the adder, where carries can be evaluated in parallel

**Benefits and drawbacks of parallelism**

Most software programming is, essentially, serial. Although functional languages express algorithms without state, most software programming is, essentially, serial. Imperative languages such as Java or C code algorithms as a series of successive statements. Of course these map well onto the underlying processor which executes them as a series of successive machine instructions. 'Parallel programming' largely refers to writing a set of largely independent threads or processes which are each made up of sequences of instructions. It is therefore not surprising that many people begin writing HDL code in a similar way. This typically results in over-complicated code and inefficient hardware.

Speed

If two or more units are able to process at the same time then the elapsed time for the whole task will be reduced. This assumes that there are no serial dependencies between the individual jobs.

An example can be seen in a superscalar processor. It is often possible to execute several instructions simultaneously; it is not possible to execute hundreds of instructions simultaneously because of dependencies.

Resources

Parallelism implies increasing the resources used. There is therefore a cost/benefit trade-off. It is not usually sensible to have many functional units which are underemployed; it is better to have a more flexible unit which is kept busy. This is the rationale for having programmable devices such as microprocessors in the first place.

Complexity

In some ways parallelism can increase the complexity of a system because there are more things happening at once.

In other ways complexity can be decreased. It is easier to design and verify small, simple units so dividing a system in a sensible way can actually make the blocks easier.

The simplification or removal of sequencing logic can help here; if a unit does only one job there its control logic is trivial and there will be no need for many of the buses and multiplexers otherwise required.

As an example, consider the ARM instruction set. Additions are used in several places:

- ADD instruction: ADD R1, R2, R2
- Address calculation: LDR R4, [R5, #&67]
- Branch offset determination: B label

Originally these all shared the same adder (for cost reasons). Nowadays they may have ‘private’ adders which don’t need to accommodate other operations. This also means that less bus multiplexing is needed, simplifying control.

Power

In some cases parallelism can be used to reduce power consumption. This is usually attributable to the ability to remove expensive speed optimization within a unit but provide an adequate overall performance using several units.

A (contrived) example could be addition: several slow but simple ripple carry adders may provide the same throughput as one larger, fast adder, and may switch less leading to a lower power dissipation.

Another power-saving technique is to employ several units running from a reduced supply voltage; the speed is roughly proportional to \( V \) whereas power roughly proportional to \( V^2 \). Thus two units at half the voltage may give the same throughput at half the power cost.
Pipelining

The concept of pipelining should be familiar by now!

Pipelines typically:

- allow an increase in clock speed
  - consequently increase throughput (providing there are not excessive dependencies)
  - increase latency
- have a small area overhead
- may save power
  - clock loading is increased …
  - … but logic switching may be reduced (by preventing glitch propagation)
- can simplify the design process
  - if used judiciously, by decomposing into ‘sensible’ blocks

They are therefore, generally, a Good Thing.

Pipelining

Pipelining can be a ‘cheap’ way to increase performance. It is cheap(ish) because it introduces parallelism without adding (much) extra logic. Clearly additional latches are required but the processing logic can often be the same as in an unpipelined version.

Pipelined logic typically has higher throughput than unpipelined logic but will always have higher latency too. It is therefore only useful for processing streams of data. The above example illustrates an imperfectly balanced pipeline where the clock cycle is dictated by the slowest stage and some time is wasted in the faster stages. This is typically because the logic will divide ‘naturally’ in certain ways which may be worth retaining for design simplicity and readability. Dividing the logic into a three-stage pipeline – and assuming this can be kept full – has yielded a better than 2x increase in throughput.

A classic pipeline example is the instruction stream in a processor. Data streams, such as are used in graphics, are another good example.

In the laboratory example it is easy to form a pipeline of about three stages (whether the first stage is part of the pipeline could be debated).

The first stage is the pending command and its parameters which can be buffered whilst a previous command is executing. Although this is not operating at the speed of the rest of the pipeline it does provide some ‘elasticity’ in the process and can help to keep the drawing engine busy by providing the next command as soon as the current one finishes.

The other two stages, as shown above, are the iterative drawing stage and the memory write operation. An obvious parallelism exists by calculating the next pixel concurrently with writing the current one.

Possible pipeline penalties

As observed above, pipelining will usually increase latency. Typically this is not an issue in streaming data applications but may be a problem where there are dependencies forming data loops, such as the registers in a processor.

Pipeline stages can be largely independent but there can be circumstances where a stage stalls. In this case it may be necessary to stall the whole pipeline, resulting in a control overhead. Introducing a global control circuit can impact performance as it has to gather all possible stall signals, OR them and rebroadcast the result within the cycle time. This may impact the cycle time, especially if pipelining communications circuits.

[An alternative strategy is discussed in the interconnection lecture.]

Often, the required throughput of a pipeline may be less than its maximum. In such circumstances a validity indicator can be used to ensure only ‘real’ data is processed.
Data parallelism

**SIMD** architecture can process more, smaller data elements simultaneously.

Interleaving can allow more time for individual access.

It is possible to interleave more than two units, of course.

Explicit parallelism

An ‘obvious’ method of increasing throughput is to do several things at once. For example, memory bandwidth can be increased by increasing the width of the memory (e.g. fetching 64 bits/cycle instead of 32). This method assumes that data flows are predictable.

For many years, high performance processors have attempted to execute more than one instruction at once. Approaches include:

- **VLIW (Very Long Instruction Word)** where several operations are specified together explicitly.
- **Superscalar** where a conventional ‘sequence’ of instructions is examined after being fetched and more than one operation is issued in a clock cycle.

An approach which is now quite common is to use **SIMD** (Single Instruction, Multiple Data) processing. At its simplest, think of a 32-bit processor needing to add a set of 8-bit quantities. Rather than doing these individually, four bytes can be packed into the datapath and added simultaneously. This requires some slight changes to the adder to prevent a carry from one byte affecting an adjacent sum but this is cheap to do. Many processors have been extended to incorporate such support including the Intel ‘x86’ architecture [MMX, SSE, …] 64- and 128-bit units and ARM’s 64-bit ‘NEON’ extensions. These extensions are intended to accelerate stream processing, such as video. Graphics Processing Units (GPUs) typically contain wide, SIMD datapaths for this reason.

Interleaving

Interleaving is an old technique often associated with memory accesses which cannot easily be pipelined. If a memory is too slow to be read in a single cycle it may be possible to have two memories, reading from each alternately and shuffling together the results. This can work for write operations and predictable reads such as code fetches.

However, let’s look at a logic example instead. Take some sequential logic – here represented by a counter – which is too slow for the clock period. Instead of counting in ones at the clock frequency, count in twos at half the clock frequency. This may necessitate using two interleaved counters which only change value on alternate cycles. This allows twice as much time for the logic to operate. (Actually slightly more than that because the register delay is not a factor in the ‘middle’ of the period.) The desired value can then be selected with a multiplexer. Assuming circumstances permit, a following latch is often a good idea in order to ensure a ‘clean’ output which changes as early as possible in the clock cycle. This will introduce another cycle of latency which may need accounting for in the design.

Here is some example Verilog code to illustrate the operation.

```verilog
always @(posedge clk) // Modulo 13 counter
begin
  En <= !En;
  if (En)
    if (count_1 > 10)
      count_1 <= count_1 - 13;
    else
      count_1 <= count_1 + 2;
  else
    if (count_2 > 10)
      count_2 <= count_2 - 13;
    else
      count_2 <= count_2 + 2;
    if (En) count <= count_2;
  else count <= count_1;
end
```
**Choices of microarchitecture**

**Basic multiplication**

Multiplication is repeated addition. To multiply A and B, simply start with 0 and add A to it B times. This gets the answer but, clearly, is slow if B is large.

'Long' multiplication multiplies one digit at a time and shifts the partial result to the appropriate digit position. This is much quicker because there are only \(\text{number of digits}\) additions.

Binary multiplication is easy because you only multiply by 0 or 1 which is an AND function.

There are many ways to implement a multiplier, some of which are sketched out here. The variety of choices is a reason why Verilog synthesizers will not simply implement the '*' operator.

**Iterative multiplier**

With an adder and an accumulator it is possible to multiply two numbers in N cycles, processing each bit of 'B' in its own cycle. This gives a latency of N cycles and a throughput of 1/N. It is cheap in hardware and is suitable for 'occasional' multiplications, for example as in a general-purpose microprocessor.

By iterating starting at the least significant bit it is possible to improve performance by early termination; when the only bits left to multiply by are zero the result will not change further and iteration can be stopped. This speeds up multiplication by small numbers.

**Pipelined multiplication**

A pipelined multiplier could perform each addition in a separate stage. This would have a latency of N cycles but give a peak throughput of 1. The advantage is that the clock rate can be much higher than if propagating the carry. This disadvantage is that more stages are required because, eventually, the last carry has to be propagated all the way. It thus has a latency of 2N cycles with a throughput of 1, but the cycles can be much shorter. Alternatively, if the cycle time is limited by other units, the carry can be propagated a few places in each cycle.

This architecture is similar to the pipelined architecture but probably a bit faster but with a bit more area cost. However it is a very regular structure.

**Systolic array multiplier**

The critical path in addition is the carry propagation. It is not necessary to propagate the carry in every addition stage, it can be latched and input to the next significant bit in the next stage, rather than the current one. The advantage of this is that the clock rate can be much higher than if propagating the carry. The disadvantage is that more stages are required because, eventually, the last carry has to be propagated all the way. It thus has a latency of 2N cycles with a throughput of 1, but the cycles can be much shorter. Alternatively, if the cycle time is limited by other units, the carry can be propagated a few places in each cycle.

This architecture is similar to the pipelined architecture but probably a bit faster but with a bit more area cost. However it is a very regular structure.

**Parallel multiplier**

It is possible to derive all the partial products in a single cycle and add them in a tree of adders. This can give a low-latency addition with quite a high throughput; in principle latency and throughput would both be 1 although the cycle time needs to be long to accommodate multiple additions. In practice such an adder would probably be pipelined and may employ some 'carry save' tricks as well. Such a unit could yield a latency of \(-\log_2(N)\) with a throughput of 1 product/cycle, the cycle length being comparable to the earlier examples. The hardware cost is significant; the low-latency could make this appropriate for a high-performance processor running 'general' code.

**Caveat**

The descriptions given here are illustrative and contain numerous simplifications. For example, it is common to produce 2-bit rather than 1-bit partial products which halves the number of subsequent additions. However they do illustrate the sort of options open to a designer and that the 'correct' choice depends on various circumstances.

**Further reading**

If interested, you could look up terms such as: Booth’s Encoding, Wallace tree, Dadda multiplier, …

---

**Example: Multiplication**

- Iterative
- Pipelined
- Parallel
- Systolic array
RTL parallelism

The two C language loops below (Bresenham) are functionally identical but formatted differently

The LH example is expressed serially – appropriate for a microprocessor with one ALU.
Neglecting the comparisons (and ‘branches’) this takes 4 or 6 cycles
The RH example is formatted to show how it could be executed on a superscalar CPU.
This takes 2 or 3 cycles assuming (at least) three ALUs

We can do better than that, if we choose!

The intention is to highlight the scope for parallel implementation.

**Code style**

The right hand column shows a C language implementation of a line algorithm. This can be divided into quite a long preamble and an iterative loop which moves the address and plots each point.

Note:
- For a line of significant size, most of the activity will be in the loop
  - This is the place where optimizations pay off
- The code has been structured so that the output is the address
  - This saves plotting points with the implicit multiplication each cycle
- The loop body has been written in a slightly obscure fashion
  - This is to ‘remove’ asymmetry in the control structure

Let’s look at some ‘simpler code’. (This example is restricted to a particular octant but that is not relevant for the moment.)

Executing one line/clock a loop would take 6 or 8 cycles.
- Some parallelism is apparent
  - e.g. the counter (c--=) is independent
- There are dependencies.
  - e.g. the if depends on a previous operation and a test

You may spot that the tests (while/if) are tests for positive which is already represented by a single bit; thus they cost ‘nothing’.

Bresenham’s Line Algorithm

Implemented in C

void line(unsigned int x0, unsigned int y0, unsigned int x1, unsigned int y1, unsigned int col, unsigned char *screen)
{
    unsigned int address;
    int dx, dy, adx, ady;
    int a1, a2, sx, sy;
    int e, c, m, n, p;
    /* Set up - written 'single assignment' for combinatorial implementation. */
    address = x0 + y0 * WIDTH;          /* Two adders, assuming fixed width = 640 */
    dx = x1 - x0;                                                        /* Adder */
    dy = y1 - y0;                                                        /* Adder */
    if (dx < 0) sx = -1;     else sx = 1;
    if (dy < 0) sy = -WIDTH; else sy = WIDTH;
    adx = abs(dx);                                                       /* Adder */
    ady = abs(dy);                                                       /* Adder */
    if (adx > ady) { a1 = sx; c = adx; m = adx; n = ady; }   /* Comparator plus four multiplexers */
    else           { a1 = sy; c = ady; m = ady; n = adx; }   /* four multiplexers */
    e = -m;
    p = n - m;
    a2 = sx + sy;                                                        /* Adder */
    /* Plot first point (really jump into loop)*/
    screen[address] = col;
    /* Iterative loop */
    while (c > 0)
    {                               /* Four adders + two multiplexers + counter */
        if ((e + 2*n) <= 0) { address = address + a1; e = e + 2*dy; }   /* Comparator plus */
        else                { address = address + a2; e = e + 2*dx; }   /* four multiplexers */
        c--;
        /* Plot */
        screen[address] = col;
    }
    return;
}
}
Parallel RTL example

- Single cycle operation: control is simple
- Precalculated increment/decrement values: some setup required
- Rapid cycle: critical path is an adder plus a multiplexer
- Requires several parallel adders
- Some calculations are speculative: wastes power
- Plot is pipelined

The slide shows a block diagram of the Bresenham’s line algorithm loop implemented in a highly parallel way. This is designed for single cycle operation.

- On the left two possible next addresses are calculated: one accounts for a horizontal or vertical step (as appropriate) the other for a diagonal step.
- At the same time the central section calculates the ‘error’ both if the ‘if’ should/should not be taken. The increment registers holds appropriately precalculated values.
- The result of one of these operations is tested to see if the ‘if’ should be taken. This is simple because the comparison is ‘> 0’ which is the sign bit in a two’s complement notation.
- The appropriate bit switches the two multiplexers to route the desired new values to the accumulator registers.
- The output address is sent to a subsequent pipeline stage, together with the pixel data (not shown).
- On the right a counter controls the iteration of the whole unit.

This unit is fast. There are two series logic blocks in the cycle but one is a multiplexer (expect to be two-gates deep) so the critical path is little more than the adder time. It is also capable of producing an address on every clock cycle. (This may be ‘overkill’ if the plotting mechanism cannot keep up!)

It is quite expensive in hardware terms: there are four adders and the counter.

It is probably not the most power-efficient design because it employs speculation; it does work that may never be needed, just in case it is.

Single cycle operation means the sequencing control logic is simple: stop/go.

Initialisation

The setup is quite complicated. In this design it was assumed that this was performed in software by a processor. This could be done in parallel, with the next line parameters being calculated and buffered whilst the current plot is iterating.

This is valid reasoning if:

- lines are expected to be reasonably long
  - a quick estimate suggests 30-40 cycles for the setup; if lines are very short then this would be the limiting factor
  - the processor cannot be doing something more useful
  - the area cost of a hardware preprocessor is … unpalatable

Another possibility would be to pass simpler parameters (there will always be some cost in writing to the I/O registers) and perform the setup locally.

- building all the setup logic combinatorially would be quick (although the critical path would be much longer than our plot loop so several clock cycles may be needed) but expensive in hardware
  - this logic would be idle most of the time
  - it may be worth it if you have a big silicon budget
  - it’s simple to design
- devising a small ‘processor’ to perform the parameter transformations
  - area efficient
  - allows CPU to do other things
  - probably fast enough to keep up with parameter loading speed
  - needs sequencing/decision control logic

Afterthought

Lines come in different lengths. If these are a mixture of long and short lengths there may be some benefit in preparing and buffering more than one set of parameters in advance to try and keep all stages busy as much as possible.

This may improve overall speed (but by how much?). This would cost some area and complexity for the FIFO.
Conclusions

- There are many factors to consider when setting out to build a system
- You probably won’t find ‘the optimum’ solution …
  - is it worth trying?
  - will the ‘goalposts’ move?
  - there are probably things you don’t know
- … but you should aim to get close

- Your design constraints may differ from project to project
  - speed is tempting but fast enough is fast enough
  - power (energy) increasingly important – more later
  - area smaller is probably better
  - simplicity improves time to market (or ‘reliability’)

Evaluation

With a synthesizable RTL design it is possible to begin to get some idea of its feasibility by synthesizing the block and checking the output statistics.

Example: line algorithm

Two implementations of the line drawing algorithm are compared below. These both use the same iterative machine for plotting pixels, the difference being that the first includes combinatorial logic to translate pixel coordinates into the appropriate parameters whereas the second relies on this being provided by software.

Synthesis was for a Xilinx FPGA as used in the lab. In both cases no particular optimization has been attempted.

<table>
<thead>
<tr>
<th>Design</th>
<th>Slices</th>
<th>F/F</th>
<th>LUT</th>
<th>Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>With init. logic</td>
<td>194</td>
<td>78</td>
<td>332</td>
<td>80 MHz</td>
</tr>
<tr>
<td>Without init. logic</td>
<td>100</td>
<td>93</td>
<td>177</td>
<td>91 MHz</td>
</tr>
</tbody>
</table>

i.e. moving the preparation to software gives a design about half the size and 14% faster.

- It is easy to generate a lot of logic using a HDL and a synthesizer.
- Is it worth it for your application?

Key

‘Slices’ are the basic logic units on these FPGAs.
Slices contain both D-type flip-flops (F/F) and logic Look-Up Tables (LUT).
(Combinatorial logic is evaluated using pre-programmed memory tables.)

There are other elements, such as RAMs and multipliers which may be used by the synthesizer … but not here.

Time to market

Microelectronics is a fast moving milieu. The release date of a product is as important as its functionality. A particular device is going to be obsolete in the near future; sales have to take place before then.

Below is a sketch suggesting how development time affects overall sales, represented by the shaded areas.

The point being, a small difference in development time can make a big difference in profit so designer productivity is a key factor in the industry.