TSM_EmbHardw

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

02 – Scheduling / Software Optimisations / Custom Instructions

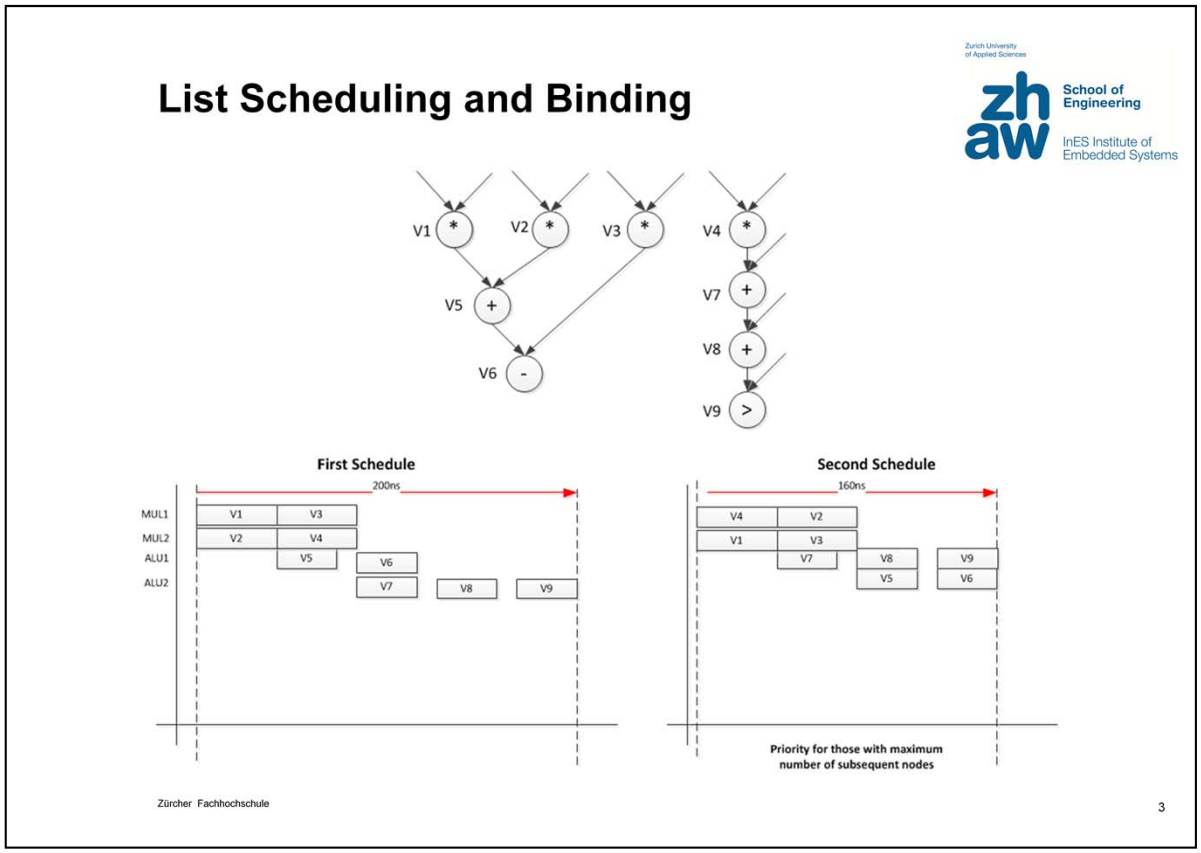
Hans Dermot Doran: Institute of Embedded Systems

V1.0	donn, 20.05.2014 New
V1.01	donn, 20.05.2014 Updated Slide 8
V1.02	donn, 23.05.2014 Added conclusions – edited slide 9
V1.03	donn, 20.05.2015 Added Voltage and RM scheduling
V1.04	donn, 20.04.2016, Added FiFO and RR scheduling, included list scheduling

What you should be able to do after today

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- The student will be able to schedule tasks using list scheduling
- The student will be able to schedule tasks using FIFO and RR scheduling
- The student will be able to schedule tasks using rate monotonic scheduling
- The student will be able to schedule tasks using critical path analysis voltage scheduling

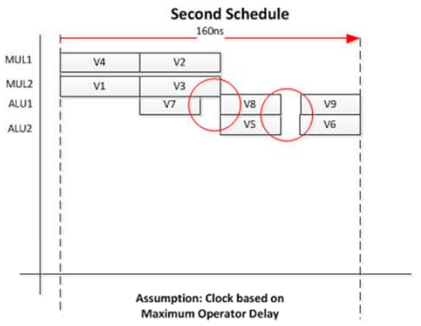


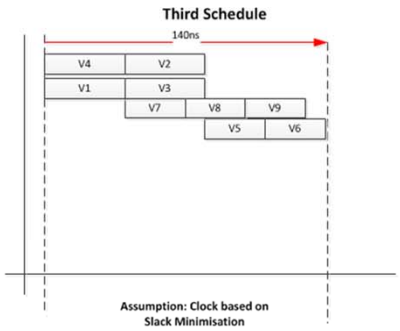
List scheduling is an algorithm typically seen in scheduling functional blocks (adders, multipliers and the like) as well as in compilers for scheduling instructions.

Viewing the DFG above we can see that there are two parallel tasks to be done within a similar time frame. Let us assume that the architecture boasts four functional units, two ALU's – of latency 30ns - and two multipliers with a latency of 40ns. We can determine the binding by scheduling. The first schedule is an ad-hoc schedule and shows some space between the task V5 and V6 – this space is known as **slack**. By prioritising the scheduling of nodes that have the largest number of following nodes we can derive the schedule on the bottom right which features a maximum latency of 160ns. This is a so-called **list schedule** which minimises latency.

Other schedules are Hu Algorithms, Force Directed Scheduling and the previously mentioned ASAP and ALAP Scheduling. List Scheduling is most frequently used. Optimising compilers also use **trace scheduling** and **software pipelining**.

List Scheduling and Clock Period





$$T = max\big(del(\vartheta_k)\big)$$

$$slack(T, \vartheta_k) = (\lceil del(\vartheta_k / T) \rceil) * T - del(\vartheta_k)$$

$$avgslack(T) = \sum_{k=1}^{|V_T|} \big(occ(\vartheta_k) * slack(T, \vartheta_k)\big) \Big/ \sum_{k=1}^{|V_T|} occ(\vartheta_k)$$

$$util(T) = 1 - avgslack(T) \Big/ T$$

Zürcher Fachhochschule

4

The schedule on the left assumes that the clock is given by the **maximum operator delay** (40ns) where:

$$functional\ units{:}((\vartheta\_k)),\ delays\ (del(\vartheta\_k))\ \text{for}\ \ Clock\ Period\ T$$

A second possibility is the **clock slack minimisation** method. The slack is defined by the second formula

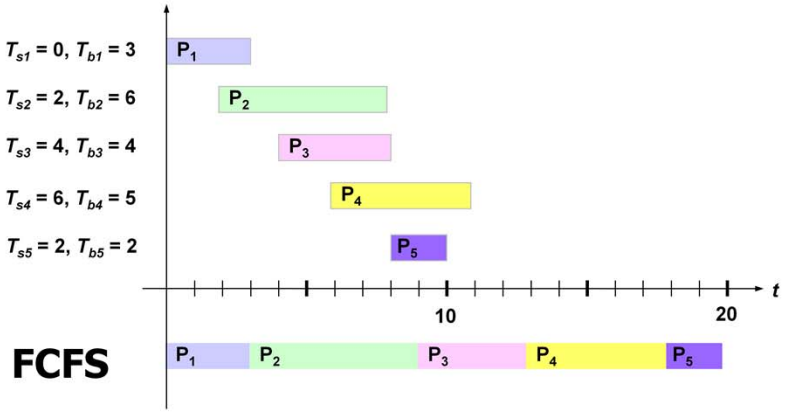
$average\ slack{:}\ avgslack,$
$number\ of\ operation\ types{:}\ |V\_T|,$
$number\ of\ operations\ of\ type\ k\ occ(\vartheta\_k)$

The idea is to maximise $util(T)$ - in this example the common divisor (zero slack) is 10ns so clocking the design at 10 ns should give the third schedule as shown above right.

Read: En-Shou Chang, Daniel D. Gajski, and Sanjiv Narayan. 1996. An optimal clock period selection method based on slack minimization criteria. *ACM Trans. Des. Autom. Electron. Syst.* 1, 3 (July 1996),

First Come First Served Scheduling

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems



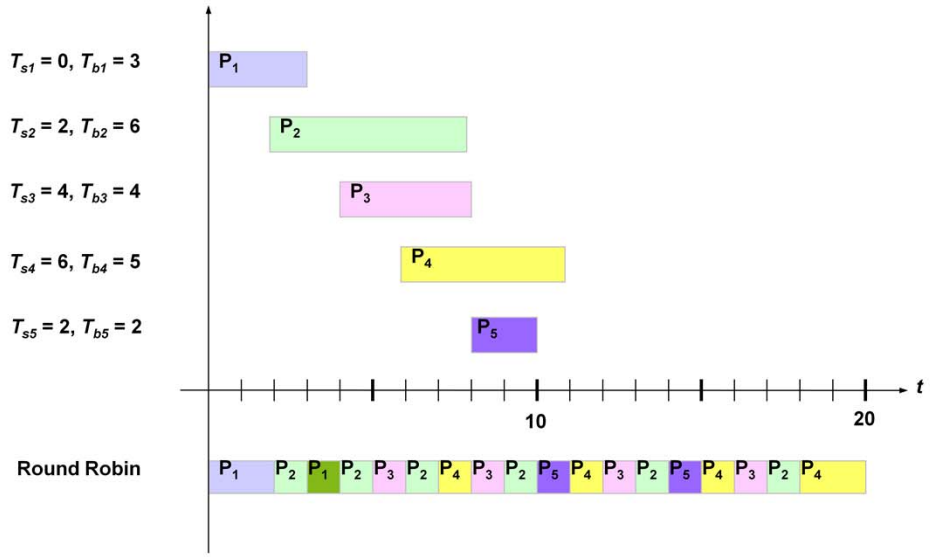
Zürcher Fachhochschule

5

First come first served is the simplest and classic scheduling algorithm. It uses a FIFO queue and is a **non-preemptive** scheduler, i.e. it doesn't interrupt tasks. It is used in operating systems – Linux has a SCHED_FIFO policy and in hardware scheduling such as bus arbitration. **Turnaround** is dependent on the run-time of all tasks in the system and their arrival time. **Throughput** is not an requirement in use cases where this scheduler is used. **Fairness** is an issue as long tasks dominate whereas **Starvation**, if a task gets no processing time because tasks of higher priority are taking up all the resource time, is not possible as there is no **Priority**, or rather the priority is implicit by arrival time. To top it all up the **Implementation** is simple.

Round Robin Scheduling

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems



Zürcher Fachhochschule

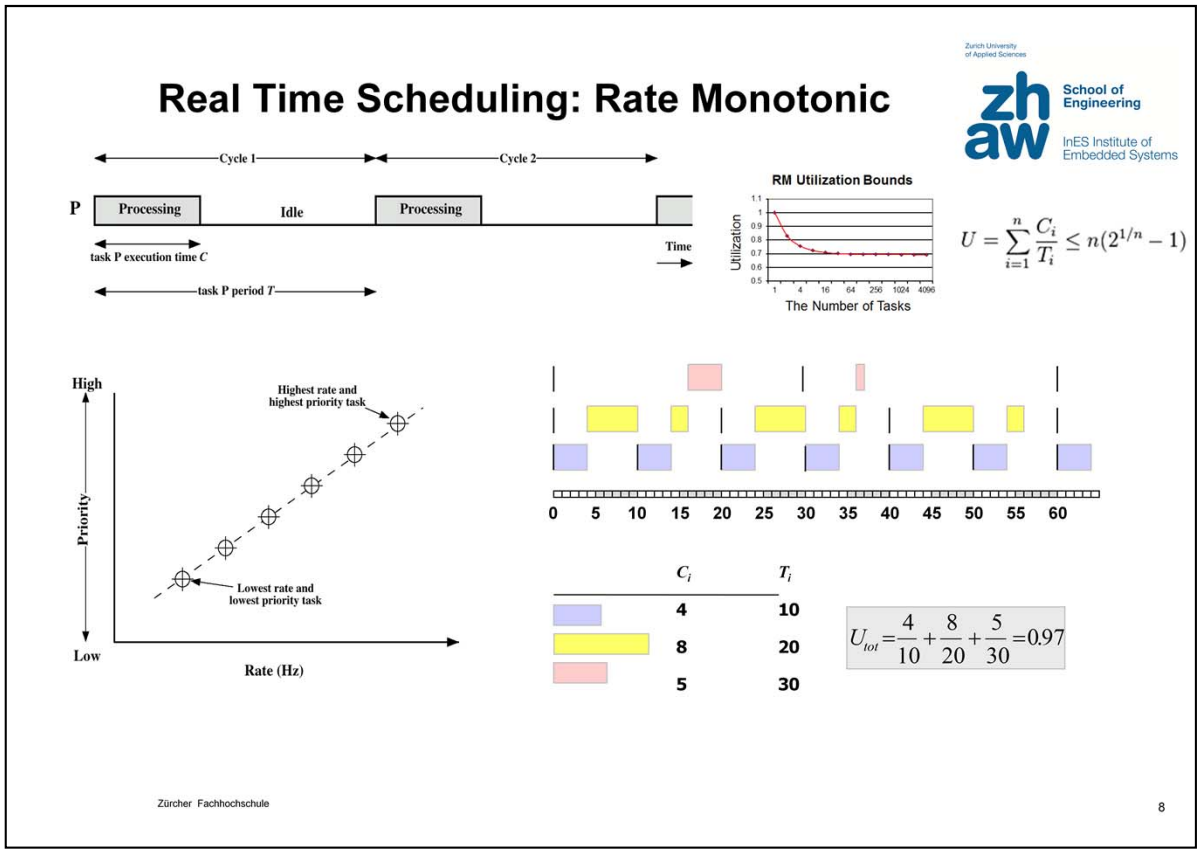
6

Round Robin is in essence a FIFO with preemption. In HW terms it represents Time Division Multiplexing. That is, execution time is divided into time-slices and each task gets to run for a timeslice. This is often to be found in simple RT operating systems but also as a HW bus arbiter.

Real Time Scheduling

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Utilisation important
- Hard Real Time scheduling
  - Deadlines may not be missed
    - Accurate WCET important
- Soft real time
  - Some deadlines may be missed
    - What happens to tasks which miss their deadlines?
    - Average Execution time more important in practice

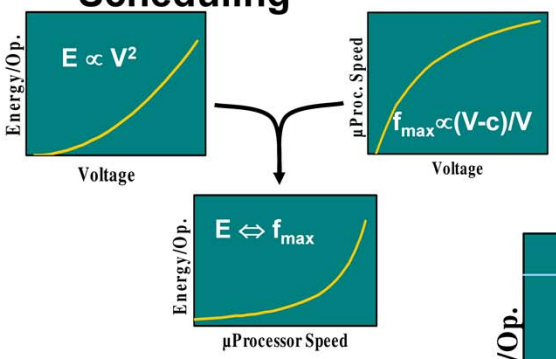


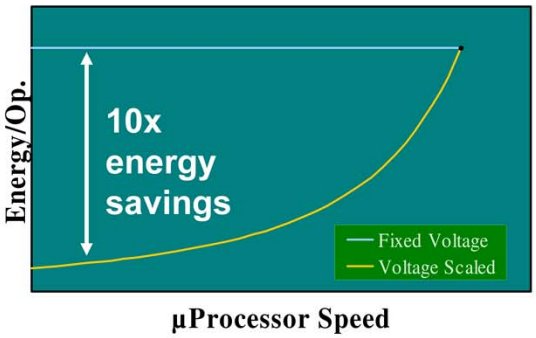
Here the basic idea is that the repetition rate represents the static priority of a task. Its a very embedded idea and can be statically applied if the number and WCET's of all tasks are known. In its simplest form it does assume the tasks are independent of each other and must be pre-emptable.

What is interesting is that a successful schedule can be proven but 0.69 is a fairly conservative estimate, in a well designed RT system 90% CPU usage can be achieved.

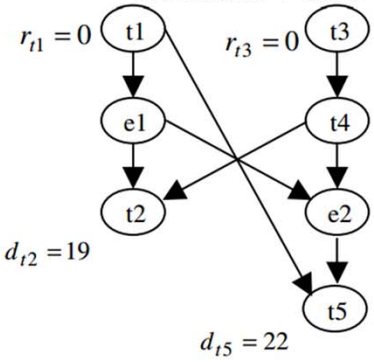
RTEMS – a military orientated RTOS supports a Rate Monotonic Manager – the manager creates a period and the programmer can attach a task to this period. The manager also collect statistics about the tasks CPU usage. http://www.rtems.com/onlinedocs/releases/rtemsdocs-4.9.0/share/rtems/html/ada_user/ada_user00322.html

Real Time Scheduling: Voltage Scheduling





Real Time Scheduling: Voltage Scheduling Critical Path



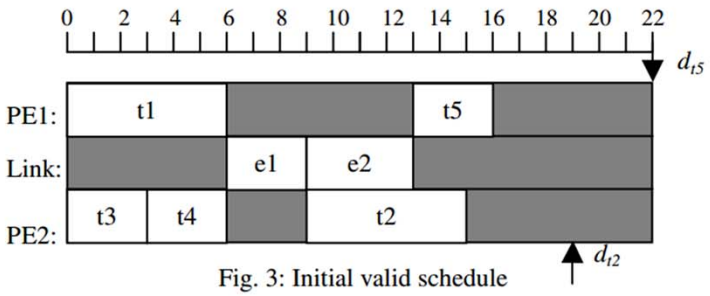


Fig. 3: Initial valid schedule

$$scale_j = (d_{destination} - r_{source}) / \sum_i wst\_exec_i$$

$$task\_scale_j = ((d_{destination} - r_{source} - \sum_i wst\_exec_i) / \sum_{i\ on\ processor} wst\_exec_i) + 1$$

Real Time Scheduling: Voltage Scheduling Critical Path

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

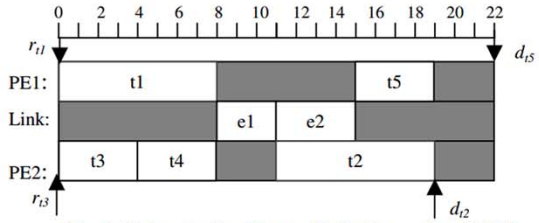


Fig. 6: Task execution times multiplied by a ratio of 16/12

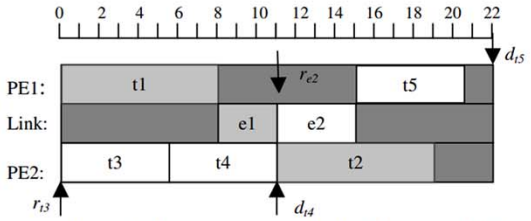


Fig. 7: Task execution times multiplied by a ratio of 11/8 (lightly shaded regions represent deleted events)

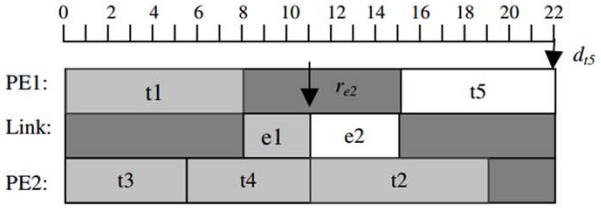


Fig. 8: Task execution times multiplied by a ratio of 7/5.5

See also:

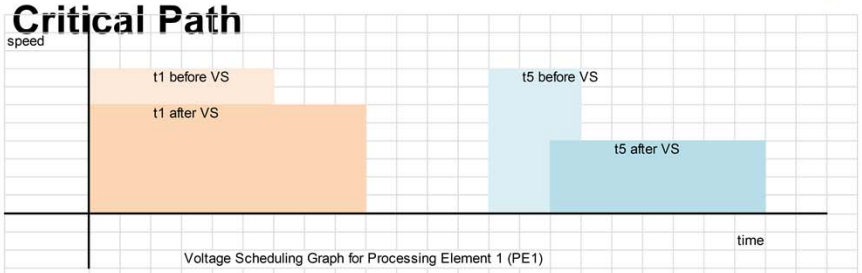
Shen, C.; Ramamritham, K.; Stankovic, J.A., "Resource reclaiming in multiprocessor real-time systems," *Parallel and Distributed Systems, IEEE Transactions on* , vol.4, no.4, pp.382,397, April 1993

http://www.cse.nd.edu/~shu/research/papers/aspdac03-1.pdf

http://www3.nd.edu/~cpoellab/teaching/cse40463/shin.pdf

Yokoyama, T.; Gang Zeng; Tomiyama, H.; Takada, H., "Heuristics for Static Voltage Scheduling Algorithms on Battery-Powered DVS Systems," *Embedded Software and Systems, 2009. ICESS '09. International Conference on* , vol., no., pp.265,272, 25-27 May 2009

Real Time Scheduling: Voltage Scheduling
Critical Path



Voltage Scheduling Graph for Processing Element 1 (PE1)

Tasks	t1	t2	t3	t4	t5	Normalised Power Consumption
Speed reduction ratio	8/6	8/6	5.5/3	5.5/3	7/3	0.5856

- Algorithm supports heterogeneous systems
- Also supports mixed systems
  - F.i. processor and invariant ASIC
- Processor support?
  - Voltage reduction in discrete steps

The results of the scheduling algorithm can be viewed against processor speed for a better appreciation of how the times and voltages (speeds) are affected.

Wise Words of Wisdom

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- *Premature optimization is the root of all evil* -- Donald Knuth
- **Profile first, optimise later.**

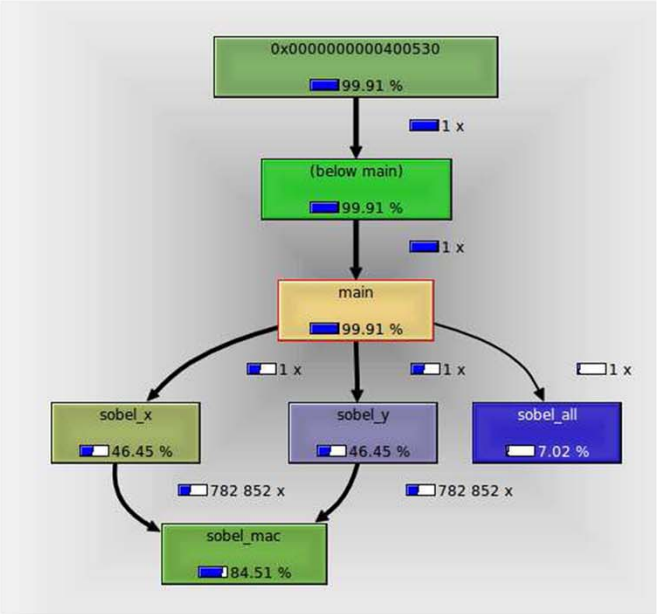
What you should be able to do after today

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- The student shall be able to understand and apply software optimisations such as loop unrolling and inlining.
- The student will be able to explain the rough structure of a compiler and where which optimisations are performed.
- The student will understand how to apply algorithmic optimisations
- The student will understand the concept and realisation of custom instructions

A Measure of Performance

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems



Zürcher Fachhochschule

15

By measuring the time spent in each sub-routine we get some, not a precise but some, measure of the relative times spent in each algorithmic function. This activity is called **profiling**: this can be achieved by various tools including *gprof* from the GNU project and *valgrind/kcachegrind*

Above we see that the processor spends 99.91% of its time in the function main, 92.9% in either sobel_x or sobel_y and 84.51% of its time in sobel_mac. Equally , sobel_x , sobel_y and sobel_all are called once each whereas sobel_mac is called a whopping 782’852 times by each of sobel_x and sobel_y. – one would imagine that there is ample scope for optimisation here.

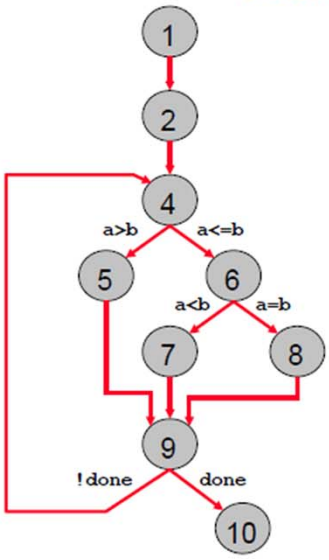
This is a typical industry practice measurement of optimisation potential here – there are some formal aspects we should consider. The first is we assume the implementation works correctly. Secondly, we are also possibly cross-profiling here, whilst we get a feel for where time is spent (ca. 1.5 million function calls is quite impressive) the actual computing time involved in making all those calls is highly dependent on the architecture the code runs on. In other words a micro-architecture analysis is required to definitively determine the ration of time spent either calling functions or performing functions.

Example: push all/pop all is generally faster than push a, push b … or a multiplication on some machines may be a fixed cycle-time operation or dependent on the input data.

One can however say that the quality of profiling results is directly dependent on the quality of the input data – garbage in -> garbage out.

Formal Background to Profiling
Control Flow Graph

```
   what_is_this {
1     read (a,b);
2     done = FALSE;
3     repeat {
4       if (a>b)
5          a = a-b;
6       elseif (b>a)
7          b = b-a;
8       else done = TRUE;
9     } until done;
10    write (a);
   }
```



From: http://www.tik.ee.ethz.ch/education/lectures/hswcd/slides/4_Mapping.pdf

Zürcher Fachhochschule

16

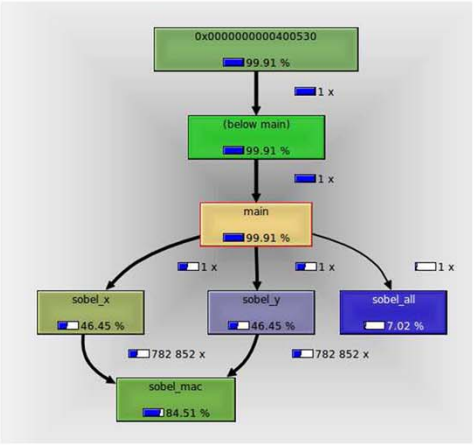
We can also do this analysis formally and by hand. By dividing up a program into basic blocks: A **basic block** is a sequence of instructions where the control flow enters at the beginning and exits at the end, without stopping in-between or branching (except at the end). When we know our basic blocks we also know our **control flow graph**. When we know this and by code inspection we can determine the **Worst Case Execution Time** (WCET) Where ci is the worst case execution time and xi is the number of times the code is executed.

$$WCET = \sum_{i=1}^{N} c_i \cdot x_i$$

This can be achieved by hand or in practical situations is simply automated by inserting counters (will slow the real-time execution of the code down) – such as done by *gprof*.

Optimisation using Profiling Information

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems



function	CPU cycles	CPU cycles/pixel
main	25	0
sobel_x	9401911	12
sobel_y	9401911	12
sobel_all	15728648	20
sobel_mac	189450184	242
total	223982679	286

Zürcher Fachhochschule

17

sobel_mac is the obvious optimisation candidate

The sobel_mac implementation

```
short sobel_mac( unsigned char *pixels,
                 int x,
                 int y,
                 const char *filter,
                 unsigned int  width ) {
   int dy;
   short result = 0;

   for (dy = -1 ; dy < 2 ; dy++) {
      for (dx = -1 ; dx < 2 ; dx++) {
         result += filter[(dy+1)*3+(dx+1)]*
                   pixels[(y+dy)*width+(x+dx)];
      }
   }

   return result;
}
```

The sobel_mac implementation

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

```
                                                        cpu cycles:
            R0 =  0          ;result                        1
            R1 =  -1         ;dy                            1
loop_y:     R2 =  -1         ;dx                            1
loop_x:     R3 =  R1+1       ;R3=(dy+1)*3+(dx+1)            1
            R3 *= 3                                         1
            R3 += R2                                        1
            R3 ++                                           1
            R3 += &filter                                   1
            LOAD R4,[R3]     ;R4=filter[(dy+1)*3+(dx+1)]  min. 1
            R3 = Y           ;R3=(y+dy)*width+(x+dx)        1
            R3 += R1                                        1
            R3 *= width                                     1
            R3 += X                                         1
            R3 += R2                                        1
            R3 += &pixels                                   1
            LOAD R5,[R3]     ;R5=pixels[(y+dy)*width+(x+dx)] min. 1
            R5 *= R4         ;R5*=filter[(dy+1)*3+(dx+1)]   1
            R0 += R5         ;result += R5                  1
            R2 ++            ;dx++                          1
            CMP R2,2                                        1
            JUMPC loop_x     ;dx < 2                       7/1
            R1 ++            ;dy++                          1
            CMP R1,2                                        1
            JUMPC loop_y     ;dy < 2                       7/1
```

Zürcher Fachhochschule

19

The calculation result += filter[(dy+1)*3+(dx+1)]*pixels[(y+dy)*width+(x+dx)] requires a total of 15 cpu cycles.

The dx-loop overhead is 9 cycles if dx < 2, and 3 cycles otherwise.

The dy-loop overhead is 10 cycles if dy < 2, and 4 cycles otherwise.

Hence the inner loop takes 15 + 9 = 24 cycles when dx 2 {−1, 0} and 15 + 3 = 18 cycles when dx = 1 means a total of 24 + 24 + 18 = 66 cycles, of which 9 + 9 + 3 = 21 cycles for loop overhead (33%!).

The outer loop takes 66 + 10 = 76 cycles when dy 2 {−1, 0} and 66 + 4 = 70 cycles when dy = 1 means a total of 77 + 77 + 70 = 224 cycles, of which 10 + 10 + 4 = 24 cycles for loop overhead (11%!).

Idea: Lets start by taking away the inner loop. -> **Loop Unrolling**

This saves 3*21 = 63 cycles; the code will run therefore 224/224−63 = 1.4× faster!

And it is effective as:

- We do not require much knowledge about the algorithm.
- We save overhead cycles

However, there are some problems:

- The effect depends on the number of cycles spend in the body of the loop.
- Only works when we have a perfect loop (read no data-depended exit of the loop).

Let’s look to the result:

The Unrolled Loop

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

```
short sobel_mac( unsigned char *pixels,
                 int x,
                 int y,
                 const char *filter,
                 unsigned int  width ) {
   int dx,dy;
   short result = 0;

   for (dy = -1 ; dy < 2 ; dy++) {
      result += filter[(dy+1)*3+0]*pixels[(y+dy)*width+(x-1)];
      result += filter[(dy+1)*3+1]*pixels[(y+dy)*width+(x+0)];
      result += filter[(dy+1)*3+2]*pixels[(y+dy)*width+(x+1)];
   }

   return result;
}
```

	Original code		Loop unrolled code	
Function	cycles	cycles/pixel	cycles	cycles/pixel
sobel_mac	189450184	242	98639352	126

Zürcher Fachhochschule

20

Looks good, the function is speeded up a factor of almost 2×

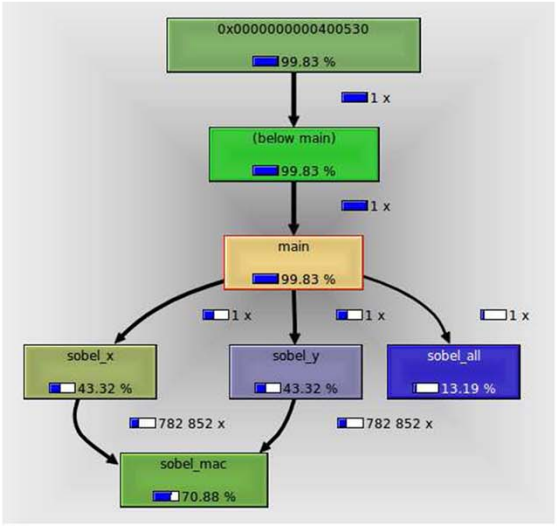
The complete application benefits from a speed-up of 1.68×

Let's see what happens if we also unroll the outer loop...

This should give us another speed-up of (224−63) / (224−63−24) = 1.18×

Re-Profiling with Unrolled Loops

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems



function	CPU cycles	CPU cycles/pixel	speedup
main	25	0	1×
sobel_x	9401911	12	1×
sobel_y	9401911	12	1×
sobel_all	15728648	20	1×
sobel_mac	84548016	108	2.24×
total	119080511	152	1.88×

Zürcher Fachhochschule

21

So the profile has changed considerably: Although sobel_all requires more clock cycles a quick look at the interface sobel_mac shows some features that are a direct result of the function having to serve two callers.

So lets have a closer look at sobel_x and sobel_y

Loop Unrolling – the negatives

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- We have unrolled a loop in a function
  - The profiler can measure the time spent in the function
  - In a longer piece of code the profiler cannot distinguish the code and hence other possible optimisations may be lost
- Not always good for efficient use of cache
  - Loop Fission -> make two loops of good data locality out of one loop

Other Loop Optimisations

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- **Loop Unrolling seeks to avoid loop penalties**
  - Loop Jamming (Loop Fusion) -> combines two loops
  - Loop Peeling -> takes first or last iterations out of loops to simplify.

Original Code	Transformed Code

`int p = 10;`
`for (i=0; i<N; ++i)`
`{`
`  y[i] = x[i] + x[p];`
`  p = i;`
`}`
	`y[0] = x[0] + x[10];`
	`for (i=1; i<N; ++i)`
	`{`
	`  y[i] = x[i] + x[i-1];`
	`}`

  - Loop Hoisting -> takes invariant code out of the loop

```
void function(bool invariantCondition)
{
  for(int i = 0; i < 100; ++i)
  {
        if(invariantCondition)
          doSomething;
        else
          doSomethingElse;
  }
```

```
// hoist condition out
if(invariantCondition)
{
      for(int i = 0; i < 100; ++i)
        doSomething;
}
else
{
      for(int i = 0; i < 100; ++i)
        doSomethingElse;
}
```


Loop peeling example from: Embedded and Networking Systems: Design, Software, and Implementation, Gul N. Khan, Krzysztof Iniewski, ISBN 9781466590656. via google books.

TABLE 4.6
Effects of Loop Peeling Transformation on Energy and Power Consumption

	Original	Transformed	%
Execution cycles	2808	2485	−11.5
Power (W)	1.034	1.006	−2.78
Energy (mJ)	0.0029	0.0025	−13.97
IPCs	0.919	0.962	4.6
Memory references	802	499	−37.78

Loop hoisting exxample from: http://www.gamedev.net/topic/620853-loop-hoisting-optimization-technique/

Next Optimisations

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

```
const char gx_array[9] = {-1,0,1,
                          -2,0,2,
                          -1,0,1};

void sobel_x( unsigned char *source,
              short *destination,
              unsigned int width,
              unsigned int height ) {
   int x,y;

   for (y = 1 ; y < (height-1) ; y++) {
      for (x = 1 ; x < (width-1) ; x++) {
         destination[y*width+x] = sobel_mac( source,
                                             x,
                                             y,
                                             gx_array,
                                             width );
      }
   }
}
```

Zür

24

By looking at the sobel x routine we can observe that gx_array contains 3 zeros. Hence we perform 3 out of 9 calculations too many. If we move the sobel mac routine inside the sobel x routine we could expect a speed-up of at least 9/9−3 = 1.5×

We call this method **in-lining**

Inlining

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

```
void sobel_x( unsigned char *source,
              short *destination,
              unsigned int width,
              unsigned int height ) {
   int x,y;
   register short result;

   for (y = 1 ; y < (height-1) ; y++) {
      for (x = 1 ; x < (width-1) ; x++) {

         result  = gx_array[0]*source[(y-1)*width+(x-1)];
         result += gx_array[1]*source[(y-1)*width+(x+0)];
         result += gx_array[2]*source[(y-1)*width+(x+1)];
         result += gx_array[3]*source[(y+0)*width+(x-1)];
         result += gx_array[4]*source[(y+0)*width+(x+0)];
         result += gx_array[5]*source[(y+0)*width+(x+1)];
         result += gx_array[6]*source[(y+1)*width+(x-1)];
         result += gx_array[7]*source[(y+1)*width+(x+0)];
         result += gx_array[8]*source[(y+1)*width+(x+1)];

         destination[y*width+x] = result;
      }
   }
}
```

Zürcher Fachhochschule

25

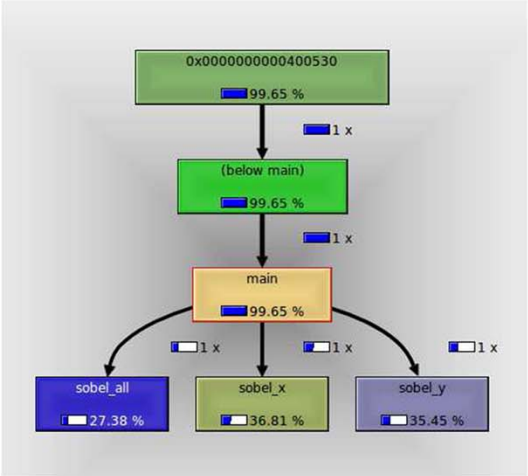
By looking at the sobel_x routine we can observe that the gx_array contains 3 zeros. Hence we perform 3 out of 9 calculations too many. If we move the sobel_mac routine inside the sobel_x routine we could expect a speed-up of at least 9/9−3 = 1.5×

We also can expect to save significantly on the calling and return overhead introduced by using a function.

We call this method **in-lining**

What trade off have we made here?

Re-Profiling with In-lining



function	CPU cycles	CPU cycles/pixel	speedup
main	25	0	1×
sobel_x	21147747	27	4.9×
sobel_y	20364895	26	5.1×
sobel_all	15728648	20	1×
total	57241315	73	3.91×

Zürcher Fachhochschule

26

The profile now looks quite good with the time being spread over the three functions. We could say there are no more cheap optimisations to be had – but we shall look at formally what we have done.

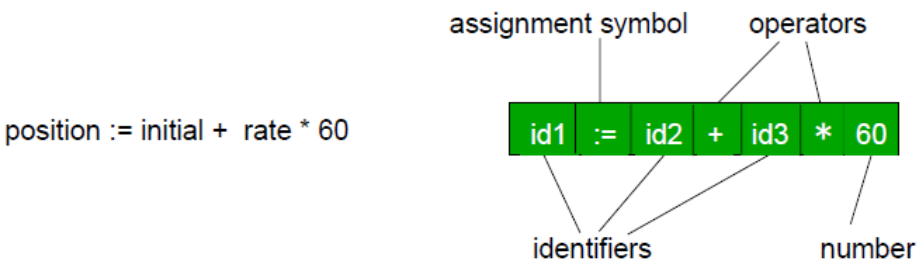
- **Some complier theory.**

Most of the diagrams from : Introduction to Compilers - Christian Plessl http://homepages.uni-paderborn.de/plessl/lectures/2011-Codesign/slides/02-Compiler.pdf – the slides aren't to be recomended – for a better overview try: http://tinf2.vub.ac.be/~dvermeir/courses/compilers/compilers.pdf which is a bit formal, or a book.

A compiler performs two main operations whilst operating on a piece of source code. It performs

Analysis: - which consists of lexical analysis, syntatic analysis and semantic analysis

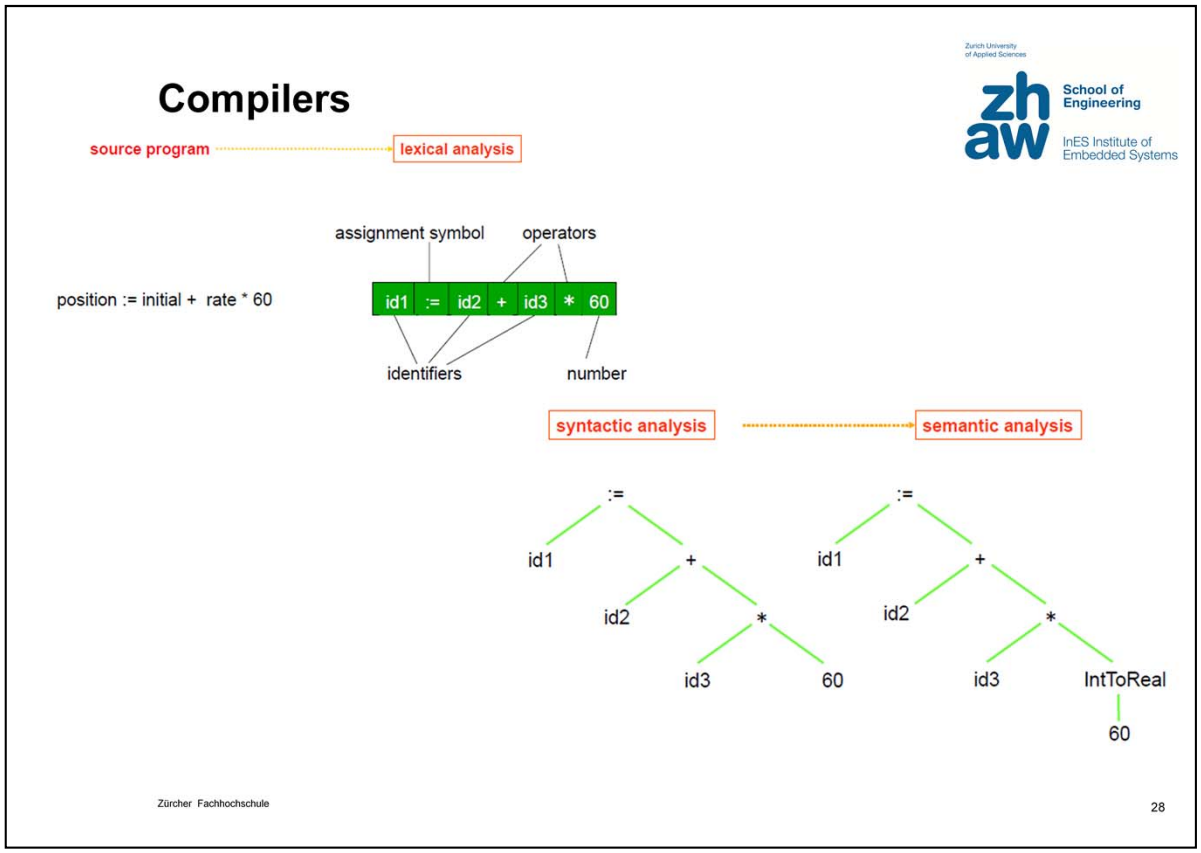
Synthesis: - which consists of intermediate code generation, code optimisation and code generation.



Optimisations and Compilers

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- **GNU Compiler offers, amongst others:**
  - *-funroll-loops* / *-funroll-all-loops*
  - *-finline-small-functions (-O2 Optimisation switch)*
    - So is: *-fdelete-null-pointer-checks !!!*

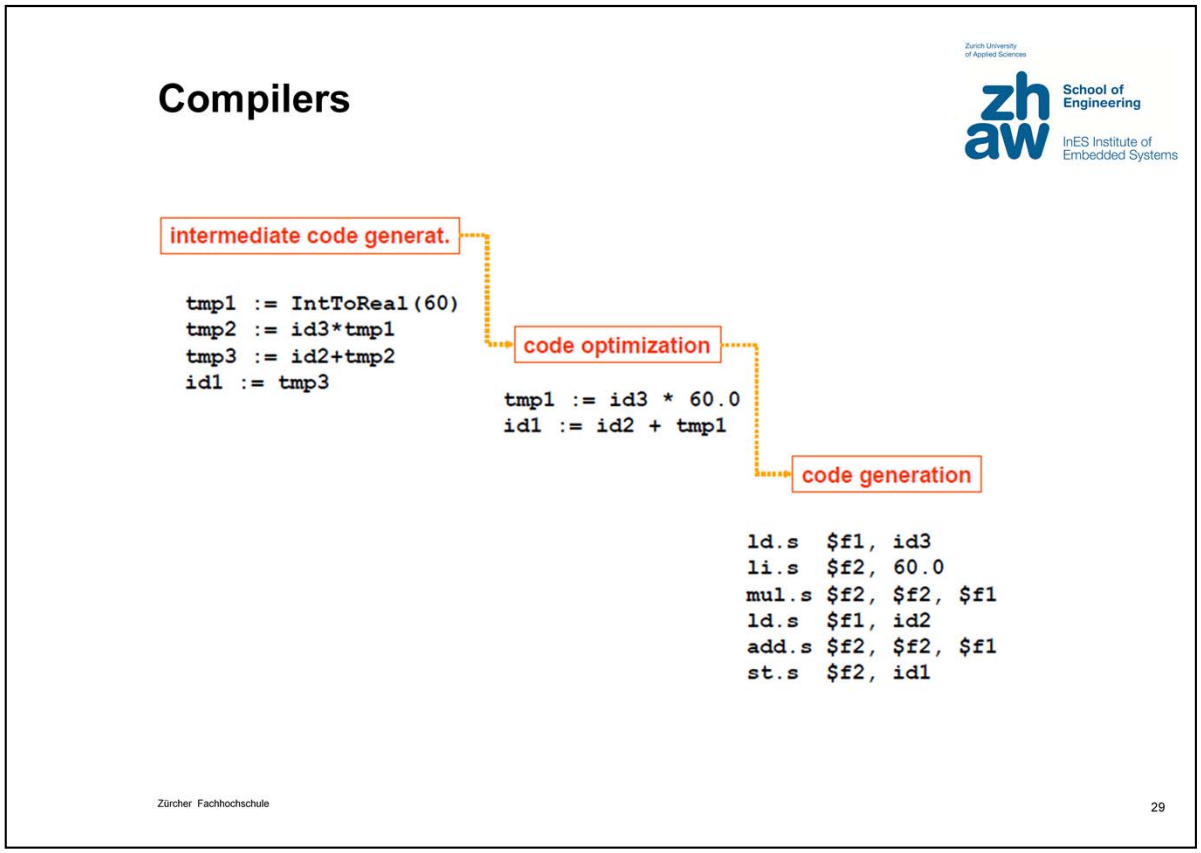


What the compiler then does is to generate so called degenerated control graphs, that is each node is a basic block rather than an individual instruction. It converts this graph into intermediate, machine independent, code. It then attempts to optimise this code before generating machine dependent code. The machine dependent code is converted by allocation, binding and scheduling.

Allocation – the components (registers ALU etc are generally fixed and the methodology doesn't leave much room for differing interpretations.

Binding - is the process of determining which variables should be held in registers or which operations happen to be in ALU's, it also is the process of instruction selection.

Scheduling – is the process of sequencing instructions according to some criteria, generally the minimum number of instructions for a given number of registers.



This code is then further optimised using machine-dependent knowledge.

So what compiler optimisations we can expect? Taking Slide 20 as an example: There are nine multiplications in the base block. Since gx_array is defined as const, by the semantical analysis stage, array accesses will have been replaced by the actual numbers (-1, 0 or 1). By the intermediate code optimisation stage the terms with multiplications by 0 will have been eliminated and the multiplications by 1 will also have been dropped. By the code generation optimisation phase a decision will have been made as to whether the *2 multiplications are to be converted into a left shift and how the -1 multiplications are to be dealt with. This is an example of local optimisations. Since in the original code the sobel_mac hid the 0 multiplications a compiler could not have made this local optimisations. Finally some compilers recommend writing for loops decrementing to zero dependent on the architectures

Function calls are an area of global optimisation – in some compilers parameter passing is cheaper if it is done via registers rather than via the stack. This statement is however often dependent on the number of parameters you wish to pass, 4 sometimes being the critical number. Knowing this means you could write code designed to place parameters used in registers where they are used from there onwards.

Loop unrolling and in-lining are generally command-line options - sometimes they are available on their own and sometimes they are bundled together with other optimisations – The lesson from this chapter is that you should study what optimisations your compiler supports.

Algorithmic Optimisations

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

```
void rgb_to_grayscale( int width,
                       int height,
                       const unsigned int *rgb_source,
                       unsigned int *grayscale_destination) {
   int loop;
   unsigned int temp;
   unsigned int grayscale;

   for (loop = 0 ; loop < width*height ; loop++) {
      temp = rgb_source[loop]&0x3F; // red value
      grayscale = (temp*30)/100;
      temp = (rgb_source[loop]>>8)&0x3F; // green value
      grayscale += (temp*59)/100;
      temp = (rgb_source[loop]>>16)&0x3F; // blue value
      grayscale += (temp*11)/100;
      grayscale_destination[loop] = grayscale|
                                    (grayscale<<8)|
                                    (grayscale<<16);
   }
}
```

Zürcher Fachhochschule

30

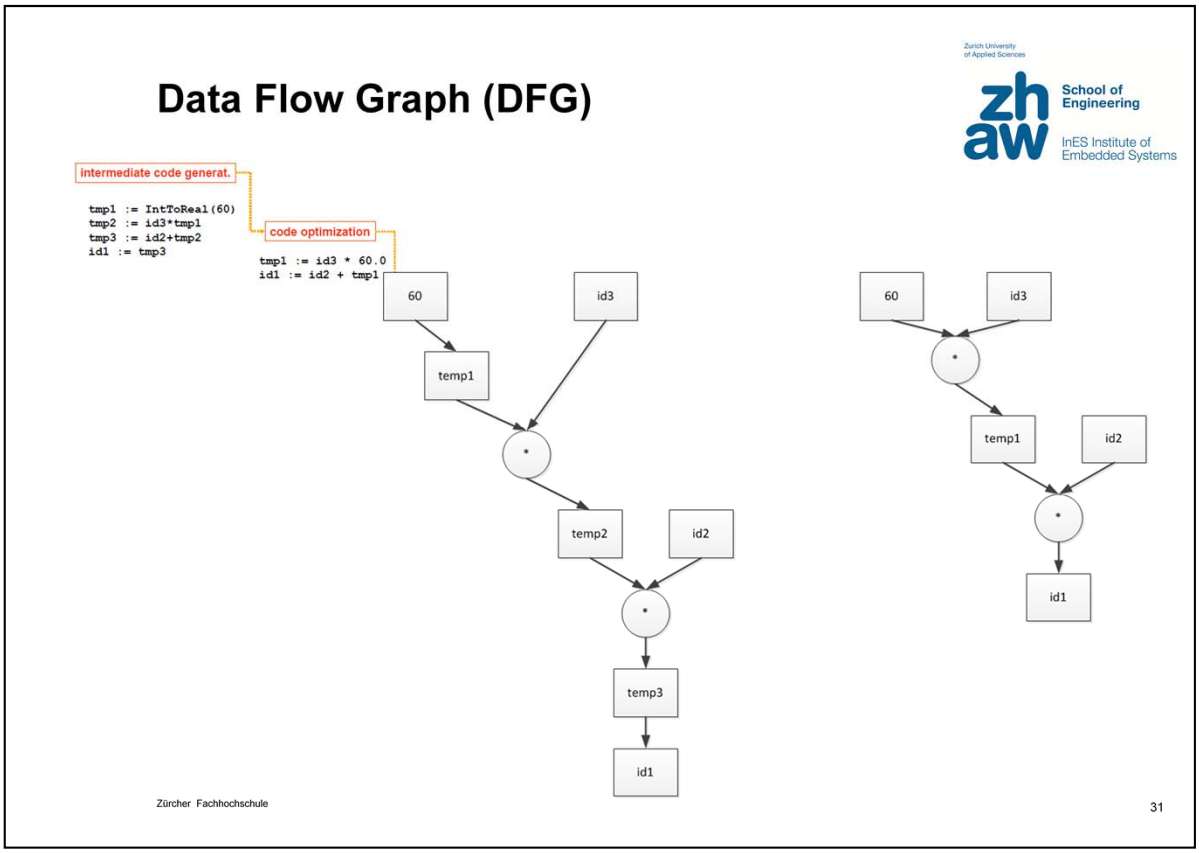
We have looked at various forms of software optimisations. With loop unrolling we have tried to eliminate what might be considered unnecessary overhead. With our lecture on cache we have seen how bus access cycles of higher latency may be reduced by the use of cache or tightly coupled memory.

Lets look at some algorithmic optimisations that might be possible: Take for an example the rgb->greyscale shown above, generally a precursor to edge detection with Sobel. What happens is that the 24 bit word from the camera (three 6 bit values for each red, green and blue) is presumed to be in memory. Each word is read, the colour bits masked and then multiplied by a factor 0.3 for red, 0.59 for green and 0.11 (adds up to 1) for blue. The grey scale is then created by adding the factorised red, green blue and written back to the picture.

When this code is profiled then, with an image of 240*320, 24652839 cpu cycles are required which corresponds to an execution time of ca. 0.5s.

The calculation takes 22579200 cycles or 294 cycles per pixel. This means that by unrolling the loop we could achieve an optimisation of (24652839 - 22579200) / 24652839 or 1.09%.

A saving of 9% isn't really worth chasing after seeing we might be able to get bigger rewards elsewhere.

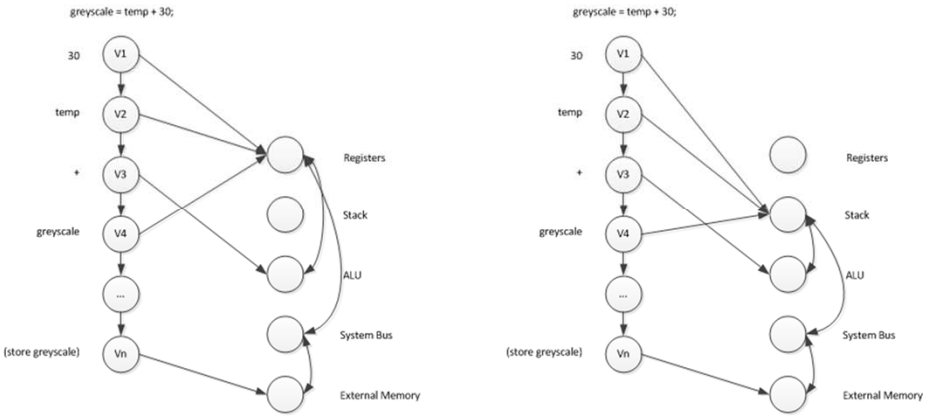


We have already met a control flow graph (CFG) – it is used in compiler technology – and indeed profiling for optimisations such as loop unrolling - for splitting up the code in to control functions and basic blocks. It is also used in verification as the code along each of the control paths should be tested. There is a pendant called a Data Flow Graph (DFG) – whereas the CFG seeks to capture the program control structures the DFG seeks to show how the data flows through the system, system being either hardware or software. It shows the data dependencies between a number of functions.

They can be used in compiler design for analysing basic blocks. we see on the intermediate code there are 4 lines of code as shown above that are then reduced to the two lines shown under code optimisation. When these are graphed the two code sets look like the graphs as shown above. Operations are generally (but not exclusively – there is no standard) in circles whereas data is in rectangles.

Note in the second graph there is scheduling information as well – if at time t= 0, the operation * takes place then at time t = 1 when the addition takes place the value id2 must be known. In a As Soon As Possible (**ASAP**) scheduling id2 would be required to be available before t= 0; in a As Late As Possible (**ALAP**) schedule id2 is only needed to be known before t = 1. The flexibility in this schedule is taken advantage of by compilers to schedule operations.

Data Flow Graph (DFG) and Partitioning/Mapping

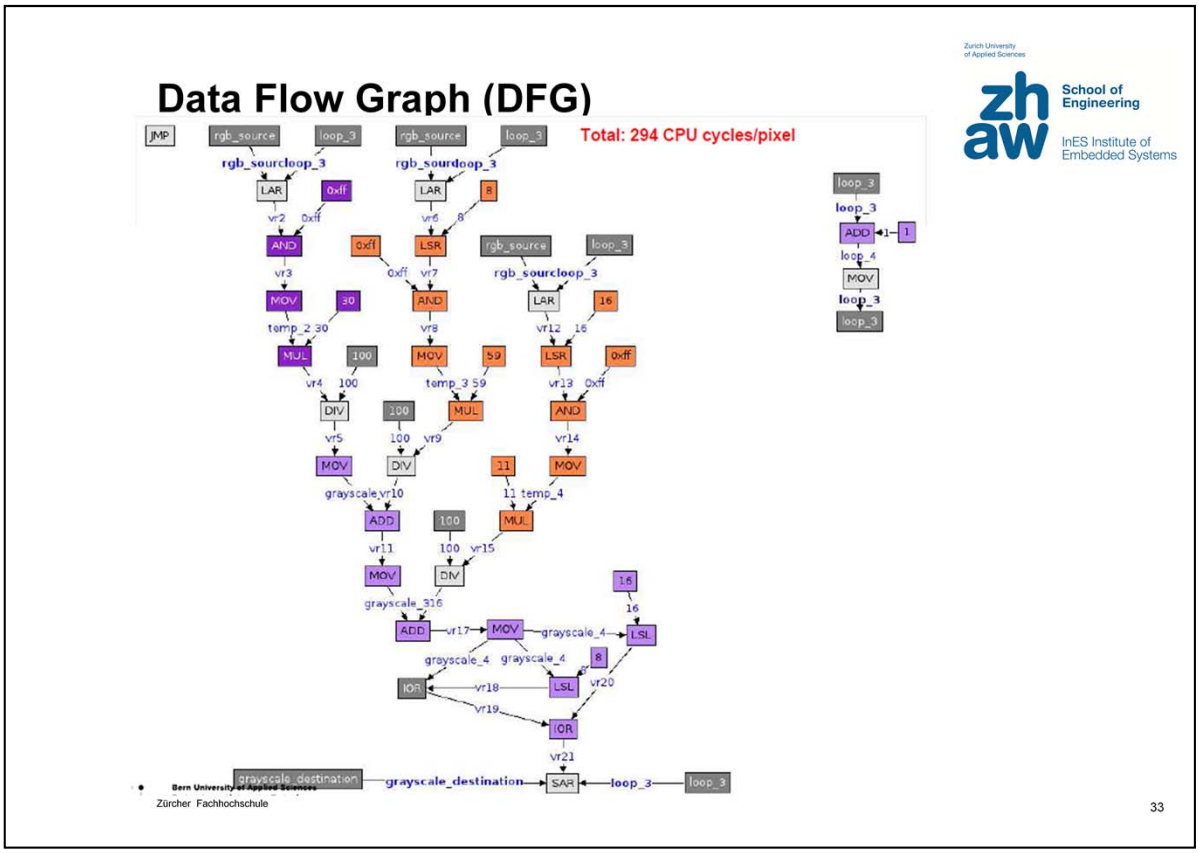


Also here we can use the data flow graph to bind the various elements of a program to processor units.

The processing units have been already allocated by design. i.e. the processor has 1 ALU, a stack (or a pointer), an instruction set and some registers.

The diagram above left shows the binding for a line of code on a register machine. 30 is bound to a register, temp is bound to a register, + is bound to the ALU (which fetches the operands from some register, chosen by the compiler and the associate instruction) and the result is bound to some register. The compiler will, at his time of choosing, dissociate the variable greyscale with the register and associate it with a memory location.

The diagram above right shows the binding for a line of code on a stack machine. 30 is pushed, temp is pushed, + is bound to the ALU (which fetches the operands from the stack) and the result is pushed on the stack. The compiler will, at his time of choosing, dissociate the variable greyscale with the stack and associate it with a memory location.



In this version of the Data Flow Graph there is no shape-distinguishing between operations and data itself. The graph is constructed from the assembler source code of the rgb to greyscale C code. There are two points we can make – the first is that the loop counter (loop_3) on the right takes 1 CPU cycle to happen. The second is that there are three divisions that are known to be expensive – in this particular case up to 32 clock cycles each. So the question is – can the number of divisions be reduced – and if so then this represents an **algorithmic optimisation**. Generally speaking one would have to include a check for **over-** and **underflow**, in this particular case given the small size of the data involved 6 bit * 6 bit is 12 bits and the final addition adds 2 bits so with a short the calculations could be done – there is no need to check for under or overflow – *it would however be helpful if it was documented somewhere* !!!

Resultant Code

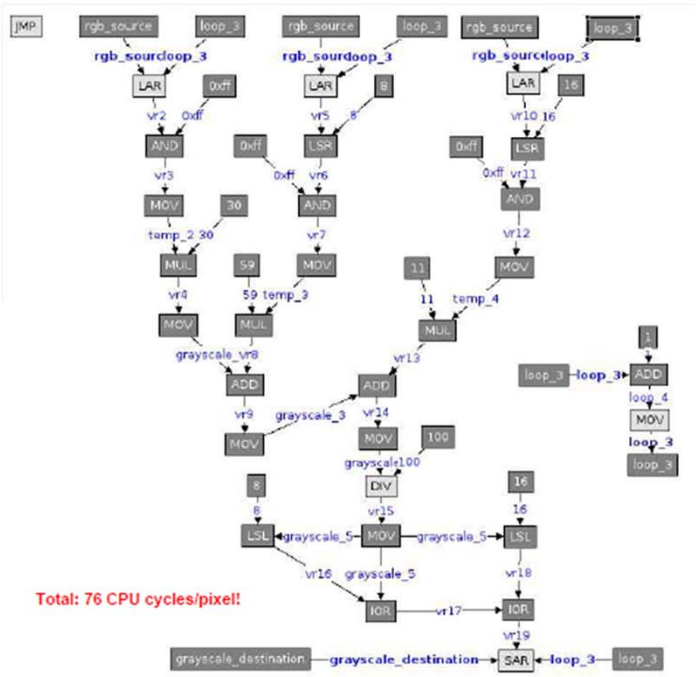
Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

```
void rgb_to_grayscale( int width,
                       int height,
                       const unsigned int *rgb_source,
                       unsigned int *grayscale_destination) {
   int loop;
   unsigned int temp;
   unsigned int grayscale;

   for (loop = 0 ; loop < width*height ; loop++) {

        temp = rgb_source[loop]&0x3F;
        grayscale = (temp*30)/100;
        temp = (rgb_source[loop]>>8)&0x3F;
        grayscale += (temp*59)/100;
        temp = (rgb_source[loop]>>16)&0x3F;
        grayscale += (temp*11)/100;

      grayscale_destination[loop] = grayscale|
                                    (grayscale<<8)|
                                    (grayscale<<16);
   }
}
```

```
temp = rgb_source[loop]&0x3F; // red value
grayscale = temp*30;
temp = (rgb_source[loop]>>8)&0x3F; // green value
grayscale += temp*59;
temp = (rgb_source[loop]>>16)&0x3F; // blue value
grayscale += temp*11;
grayscale /= 100;
```


The resultant code is shown above: Here the divisions have been removed which also represents an integer operation rather than a floating point. We are left with one division remaining.

Data Flow Graph-2



Zürch

35

In the optimised version of the algorithm the Data Flow Graph shows a reduction to 72 cpu cycles/pixel. This represents a factor 3.87 improvement.

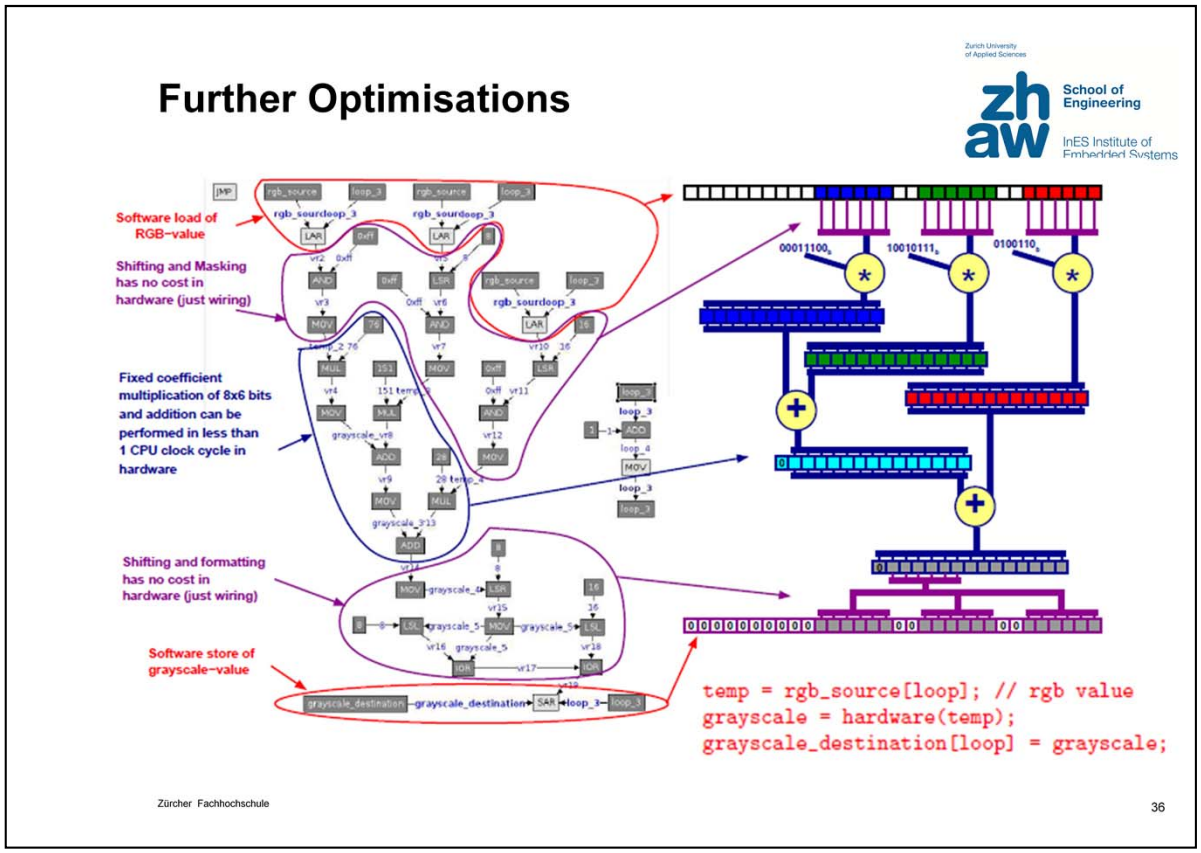
What else could one do? -> there is a further division by a constant which potentially costs 32 cycles, the constant being 100. If this could be converted into a shift by making this a power of 2 then it would make this operation into a 1 CPU cycle stage.

For example the table below shows us that by replacing the original factors by other values which are then divided by 256 -> i.e. an eight byte shift, the error induced by these approximations (compare to the error accepted by the Sobel approximation) are minimal. Therefore the multiplications can be replaced by the new values (77, 151 and 28) and the division replaced by a shift by 8.

We have now achieved a very cheap optimisation by manipulating the algorithm a little. We have 63 cpu cycles per pixel and the calculation is speeded up by a factor of 4.67.

It is important to note the lack of effort this optimisation cost us.

	Original factor	Approximated factor	Error
Red	$\frac{30}{100}$	$\frac{77}{256}$	-0.078125%
Green	$\frac{59}{100}$	$\frac{151}{256}$	0.015625%
Blue	$\frac{11}{100}$	$\frac{28}{256}$	0.0625%



By taking a look at the DFG we might be able to induce some optimisations in software but the question is whether they are worth the effort. It might be possible however to optimise the hardware: If you examine the DFG and the equivalent flow in hardware (on the right) you can see that we begin with an array containing RGB values – the load of these values into the registers is done by software.

The second phase, circled in purple, consists of shifting and masking which in hardware carries no cost, its just wiring. The beef is in the multiplication by the constants – an 8*6 (or 7*6) as well as the addition can be performed in one clock cycle by hardware, equally the multiplications can be parallelized, that is three dedicated multipliers can be built and an adder. This website provides some introductory information on multipliers in hardware: http://www.andraka.com/multipli.htm

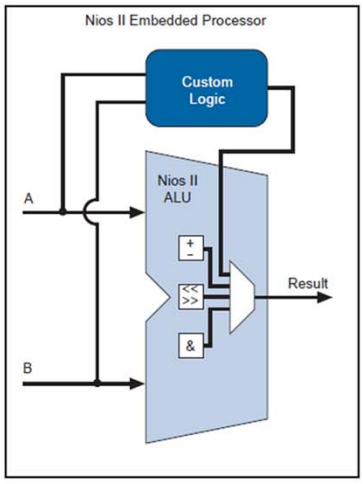
In our formal terminology - multiplication is allocated to hardware multiplier, each multiplication/colour is bound to a particular multiplier and the multipliers are scheduled so that each is finished at the same time so addition can take place.

The next operations are shifts and formatting which can be achieved in hardware and the store of the greyscale value is in software.

The perspective is to have a load a store and an increment on the loop variable so we could expect to achieve a speed up of some 73.5%. The question is how:

Well if we write the C code as shown in red above then we need some kind of connection between instruction and hardware. Luckily the IP-Processors offer such a thing in the form of a **Custom Instruction** (CI).

Combinatorial Internal Instructions HW



Type	Application	Hardware Ports
Combinational	Single clock cycle custom logic blocks.	■ dataa[31:0] ■ datab[31:0] ■ result[31:0]

Figure 1–3. Combinational Custom Instruction Block Diagram

Table 1–2. Combinational Custom Instruction Ports

Port Name	Direction	Required	Description
dataa[31:0]	Input	No	Input operand to custom instruction
datab[31:0]	Input	No	Input operand to custom instruction
result[31:0]	Output	Yes	Result of custom instruction

Figure 1–4. Combinational Custom Instruction Timing Diagram

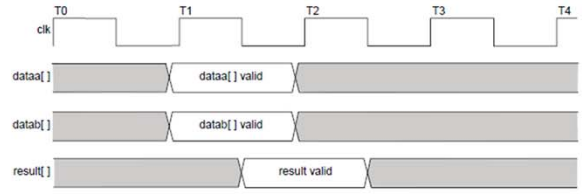
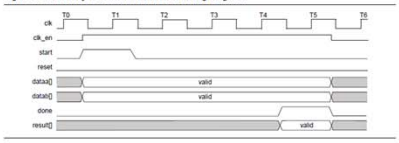


Figure 1–6. Multicycle Custom Instruction Timing Diagram



Zürcher Fachhochschule

From: NIOS custom instruction user guide, ug_nios2_custom_instruction.pdf

37

A custom instruction can be visualised as being in parallel with the NIOS ALU. That is it has connections to the data registers and writes the result into a data register, just like a standard add or subtract function. There are four types of CI. The combinational, the multi-cycle, the extended and the internal register file. The combinational – the one we are really concerned with – must be executed within one CPU cycle and accepts data on two ports and outputs data on two ports. These instructions are executed speculatively – in other words care must be taken that these don't have side effects!. Whilst the combinational instruction assume data ports dataa [31..0], datab[31..0] and result[31..0], only the result port is mandatory – if only one input port is required then dataa should be used.

The multi-cycle instruction, as the name implies, can take a fixed or variable number of cpu cycles to perform and hence have control ports, extended instructions can be used for a hardware block that allows differing functionalities and internal register file instructions allow the hardware to maintain its own set of internal registers. The timing diagram for a multi-cycle custom instruction is given as an orientation help below left.

Custom Instructions Interfacing SW

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

__builtin_custom_<*return type*>n<*parameter types*>

Example 2–3. Two Example Built-in Function Prototypes

```
void __builtin_custom_nf (int n, float dataa);
float __builtin_custom_fnp (int n, void * dataa);
```

Example 2–4. Custom Instruction Macro Usage Example

```
1. /* define void udef_macro1(float data); */
2. #define UDEF_MACRO1_N 0x00
3. #define UDEF_MACRO1(A) __builtin_custom_nf(UDEF_MACRO1_N, (A));
4. /* define float udef_macro2(void *data); */
5. #define UDEF_MACRO2_N 0x01
6. #define UDEF_MACRO2(B) __builtin_custom_fnp(UDEF_MACRO2_N, (B));
7.
8. int main (void)
9. {
10.     float a = 1.789;
11.     float b = 0.0;
12.     float *pt_a = &a;
13.
14.     UDEF_MACRO1(a);
15.     b = UDEF_MACRO2((void *)pt_a);
16.     return 0;
17. }
```

From: NIOS custom instruction user guide, ug_nios2_custom_instruction.pdf

Zürcher Fachhochschule

38

The software interfacing is fun as well: the gnu compiler has a huge array of builtin functions. These functions can be used for a multitude of things such as optimising standard library code or, for instance, code for interfacing with cache:

`void __builtin___clear_cache (char *begin, char *end)` and

`void __builtin_prefetch (const void *addr, ...)`.

Equally customised functions can be supported and the NIOS platform uses these to interface custom hardware with a custom instruction. The general form is shown in the copy of the relevant NIOS handbook where the functions return type is defined (either i or p or f) then comes n and then the parameter types that are passed to the function. Obviously, since there are only two inputs and one output to a custom instruction, there is a limited combination of possible parameter types.

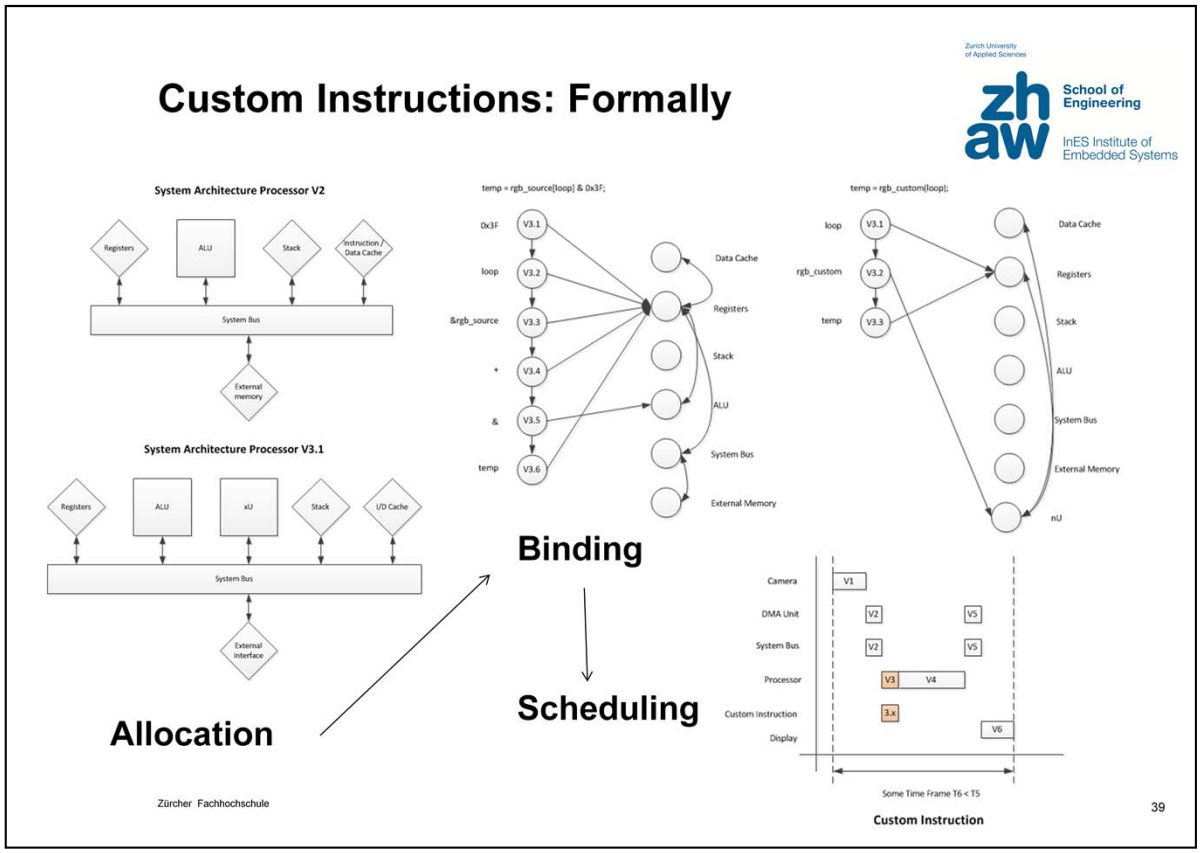
The n in the parameter signifies the custom instruction number, from 0 to 0xFF, i.e. a system can contain 256 custom instructions. The assembler custom instruction looks like:

Example 2–6. Assembly Language Call to Customer Instruction I

```
custom 0, r6, r7, r8
```

In this case the inputs are in register 7 (dataa [31…0]) and 8 (datab[31…1]) respectively and the answer is to be found in register r6 (result[31…0]).

Contrast this with the mandatory reading from Week 2!!



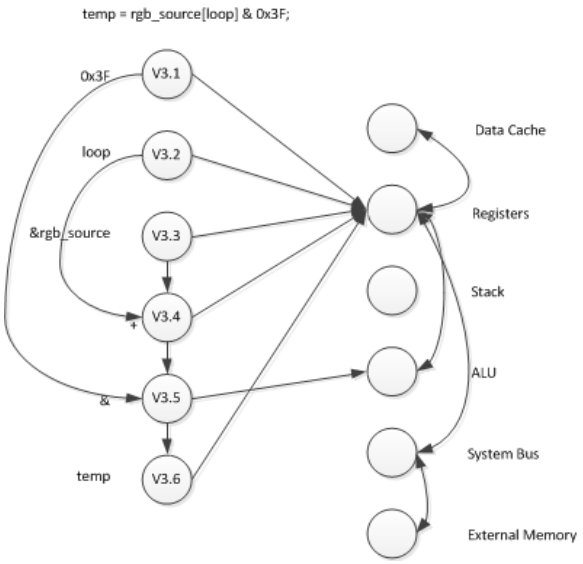
What a custom instruction does is to allocate extra hardware units to the processor. From a design which was fixed has now become variable. On the upper left we have the allocation before the custom instruction implementation. Below it the allocation after the inclusion of the custom instruction.

As an example we take the line of code as shown top right/left and show the binding to System Architecture V2.

Top right/right, as an example we show the binding to System Architecture V3.1

The resultant scheduling is shown bottom right.

Note: The DFG is shown linearly for simplicities sake! A compiler might well represent the code as shown below as there is a scheduling implication in the linear model which is not necessarily the case and which the optimiser might be able to exploit some parallelism.



Custom Instructions

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Advantage:
  - Given the CPU has already been verified
  - Only custom instruction to be verified -> unit test or in SW
- Disadvantage:
  - Has some limitations
  - Needs some courage to make first implementation