21:07 A Tourist's Guide to Altera NIOS

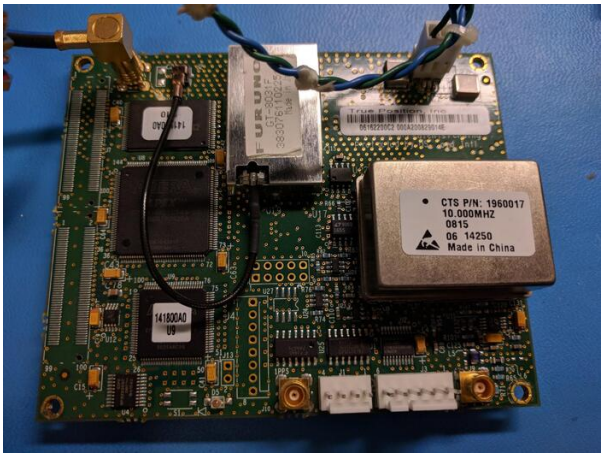
by Christopher Hewitt

Sziasztok, szomszédok!

Welcome to another installment of our series of quick-start guides for reverse engineering embedded systems. Our goal here is to get you situated with the Nios family of embedded soft processors as quickly as possible, with a minimum of fuss and formality.

Those of you who have already worked with Nios might find this to be a useful reference, while those of you new to the architecture will find that it isn't really all that strange. If you've already reverse engineered binaries for any platform, even x86, I hope that you'll soon feel right at home.

We've written this guide to broadly cover various configurations of Nios processors. These processors are generally implemented in configuration bitstreams for various Altera programmable logic devices or system on programmable chips, but may also be found in custom silicon. A minimalist configuration of Nios might be used to execute a simple control sequence out of ROM, while a complex design might make use of several fully-featured Nios processors and external memory to process a complex workload.[8] Even though Nios was quickly superseded by a complete redesign, the architecture can still be found in the occasional embedded system. Readers interested in the newer Nios II family of processors may find significant differences in the original Nios architecture and may benefit from a different introduction.



Some Historical Context

Altera introduced Nios in June of 2000 as a reconfigurable embedded design platform tailored to the company's FPGA product offerings. Building from its commercial success, Altera was quick to develop and release a successor, a 32-bit redesign called Nios II, by 2003. Having vastly improved performance and resource utilization over the original Nios platform, Altera deprecated Nios and urged developers to migrate to the new platform. After Intel acquired Altera in 2015, it became particularly difficult to find Nios-related design resources as Altera's website eventually went offline causing most references to seemingly vanish. Without having encountered a device developed during this narrow window of time it's easy to have missed out on ever seeing this architecture, though there are still some traces of Nios in the wild.

An Unexpected Rediscovery

GPS disciplined oscillators are a great way to provide a stable frequency-locked reference for test and measurement equipment found on electronics workbenches, but commercial products can be out of reach for the hobbyist on a tight budget. Fortunately, amateur radio operators have already solved this problem by repurposing the TruePosition LMU300, a nifty piece of telecommunications equipment recently decommissioned in bulk.[9] These devices were originally installed to provide caller location to emergency services in North America in accordance with federal E911 mandates. Each rackmount unit contains a separate smaller board containing a GPS receiver and a disciplined 10 MHz reference output, which can be operated independently with some modifications.

Ordinarily, the board's GPS function is initialized by another component within the chassis sending a `$PROCEED` command via RS-232. Without this command, the firmware is stuck in a loop constantly transmitting its firmware version number and device serial number. A common workaround is to have an external device send this command to the board automatically when powered on, but it's preferable for

[8] `unzip pocorgtfo21.pdf phrack6317.txt # Phrack 63:17 by Cawan`
[9] `unzip pocorgtfo21.pdf packratgps.pdf # Packrat GPS by WA2OMY and WA3YUE`

these kinds of problems to be solved in software. Since all logic is handled by an Altera APEX 20KE FPGA and MAX3000A PLD sharing 1 MB external parallel flash, that task is somewhat more challenging.

It's often a good idea to check what's stored in flash memory first. Having neither the appropriate TSOP-48 hardware programmer adapter nor the patience to wait for one to arrive in the mail presents a ripe opportunity to explore boundary scan techniques for extracting data. Simply load the relevant Altera-provided BSDL files for the FPGA and PLDs into UrJTAG and it's possible to intercept control over all I/O lines. Without access to schematics, however, it's necessary to first probe out device interconnects in order to determine which pins could be used to bit-bang data from the external flash memory. Then it's just a matter of exercising the JTAG commands SAMPLE and PRELOAD in proper sequence or, better yet, just use UrJTAG's prototype external memory bus type to automate the process. If anything goes wrong, make sure to check the boundary scan definitions for helpful hints left for hardware hackers in the distant future.

```
-- ****************************
-- *          DESIGN WARNING      *
-- ****************************

attribute DESIGN_WARNING of EP20K160ET144 : entity is

"The APEX 20KE devices support IEEE 1149.1 testing "&
"before and after device configuration; however, "&
"the devices do not support this testing during "&
"device configuration. The easiest way to avoid "&
"device configuration is to hold the nCONFIG pin low "&
"during power-up and testing.";
```

After waiting a brief eternity for data to shuffle back and forth from the boundary scan register, a complete dump of the external flash memory is finally available for analysis. One quick observation is that there are four binary chunks, each at evenly-spaced offsets and surrounded by empty space. Two of the chunks are the same size and similarly don't look particularly like any kind of program data. Since there is an FPGA on board, it's entirely reasonable to suspect that these are configuration bitstreams. The other two binary chunks, however, contain meaningful character sequences relating to flash programming and GPS operation. Even better, there's a signature near the beginning of both binaries spelling out "`Nios`." Finally, something that we can work with!

Basics of the Instruction Set

Even though Nios processors come in 16 and 32-bit variants, the instruction set is strictly 16-bit. Instructions are always half-word aligned, so the lowest bit of the Program Counter (PC) is always zero.

Data and address bus size, as well as register and ALU width, are determined by the variant used. Most instructions are shared between both variants, but the 32-bit instruction set includes extra register manipulation functions and optional support for hardware multiply. This table highlights the instruction differences between the two variants.

Opcode	32-bit Name	16-bit Name
011001	STS16S	
011010	EXT16D	ADDC
011011	MOVHI	SUBC
011101101	ST16S	
01111100100	SEXT16	
01111101000	SWAP	
01111110001	ST16D	
01111110011	FILL16	
01111110100	MSTEP	
01111110101	MUL	
10010	PFXIO	

Code targeting the 32-bit variant is easy to recognize, as jumps require an extra register load.

```
  ; Global so we can see it in dumps.
  .global nr_jumptostart

nr_jumptostart:
  PFX   %hi(_start@h)       ; 0x00
  MOVI  %g0,%lo(_start@h)   ; 0x02
.ifdef __nios32__
  PFX   %xhi(_start@h)      ; 0x04
  MOVHI %g0,%xlo(_start@h)  ; 0x06
.endif
  JMP   %g0                 ; 0x08 / 0x04 on Nios 16
  NOP                       ; 0x0a / 0x06 on Nios 16
  ; 0x0c / 0x08 on Nios 16 Signature.
  .byte 'N','i','o','s'
```

Five user-defined instructions, USR0 to USR4, facilitate accelerated data processing through additional logic placed in the hardware design. It might take some experimentation, or at least sufficient context, to determine the purpose of these types of instructions when no source is available.

Registers and Calling Convention

If you have prior experience with SPARC or other Berkeley RISC descendants, you might enjoy seeing a familiar register layout as well as sliding register windows for stack cache and the use of branch delay slots.

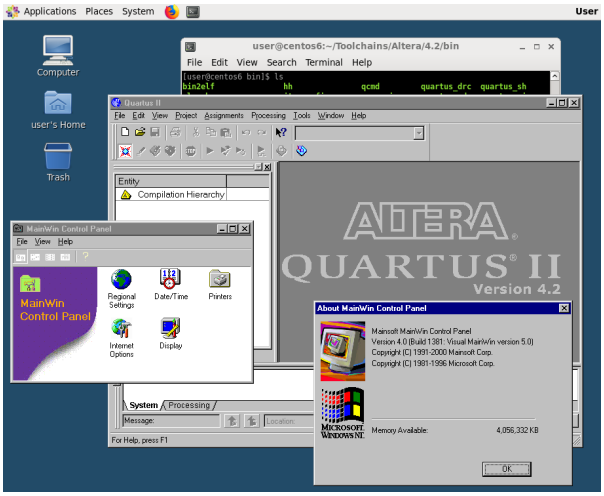
```
Inputs:  %r24 - %r31 (or %i0 - %i7)
Locals:  %r16 - %r23 (or %L0 - %L7)
Outputs:  %r8 - %r15 (or %o0 - %o7)
Globals:  %r0 -  %r7 (or %g0 - %g7)

Saved return address:   %r31 (or %i7)
Current return address: %r15 (or %o7)

Frame pointer (%fp):    %r30 (or %i6)
Stack pointer (%sp):    %r14 (or %o6)
```

A Nios processor's overall register file might span 128, 256, or 512 registers, depending on configuration. As the register window slides around, CWP is compared with the WVALID register (`%ctl2`) to determine if a register underflow or overflow has occurred, which generates an internal exception. Unless specifically disabled, Nios designs include custom exception handlers which extend the register file with extra stack memory.

Nios programs lacking any kind of register window manipulation instructions might have been compiled with the `-mflat` option. This option was intended to improve timing predictability at the expense of overall context-switching time. As a result, only a fixed 32 registers are available to the application and register contents must be saved to stack memory during interrupts since register windows are no longer available for caching.



Memory Map

A Nios processor's memory map depends entirely on how it was configured. Assuming an implementation hasn't strayed too far from one of the many original reference designs, Altera's Embedded Processor Portfolio[10] can serve as a convenient reference for correlating various peripherals to their base addresses or locating the exception vector table. Since a primary selling point of Nios (and soft processors in general) is reconfigurability, it's possible that a complete understanding will require significantly more time and effort than with a conventional hard processor.

Interrupts and Exceptions

The exception vector table can reside in either RAM or ROM at a configurable offset specified in the processor design. The table holds up to 64 exception handler addresses, depending on configuration, with each entry occupying four bytes. Exceptions can be triggered by external hardware interrupts, internal exceptions, or software instructions. The first entry in the table is a non-maskable interrupt with priority 0 only intended for use by an optional on-chip instrumentation debug module.

If the exception vector table resides in RAM and consequently generated at run-time, try tracking down the initialization code, which might resemble the following instructions:

```
  ;---------------
  ; Set up us the vector table
  ; to catch any spurious interrupt
  ; for great justice.
  ;
.if __nios_catch_irqs__
.ifdef nasys_printf_uart
  MOVIA  %o0,r_spurious_irq_handler@h
  MOVIP  %o1,nasys_vector_table
  MOVIP  %o2,64
_init_vector_table_loop:
  ST   [%o1],%o0
.ifdef __nios32__
  ADDI  %o1,4
.else
  ADDI  %o1,2
.endif
  SUBI  %o2,1
  IFRnz  %o2
  BR _init_vector_table_loop
  NOP
.endif ; nasys_printf_uart
.endif ; __nios_catch_irqs__
```

[10] `unzip pocorgtfo21.pdf nios-epp-mmap.txt # Memory maps of reference designs.`

Memory and Peripheral Access

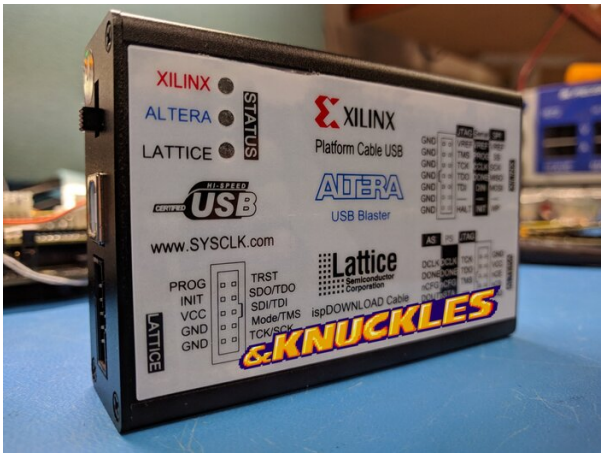
Nios has three address modes: (1) Full-width register-indirect, (2) Partial-width register-indirect, (3) and 5/16-bit immediate. Both of the register-indirect modes support an optional offset.

Nios requires the use of aligned memory accesses, so operations are performed on addresses which are multiples of two (16-bit variant) or multiples of four (32-bit variant). The lowest bit or two bits of the address are always treated as 0, respectively.

Partial-width memory reads require the combination of a full-width register-indirect read instruction with an extra `EXT`-prefixed extraction instruction. Partial-width memory writes, however, can be accomplished with a single dedicated `ST`-prefixed instruction. The additional `FILL`-prefixed instructions are helpful for meeting alignment requirements.

Disassembly

Don't worry if the Hex-Rays sales team stopped returning your phone calls. IDA Pro and other popular commercial tools don't currently support the Nios architecture anyway. Fortunately, some of the original components of the GNUPro Toolkit for Nios by Cygnus are still currently available on Sourceforge through the CDK4NIOS project. At the very least, its Nios target support for GNU `binutils` is enough to get started with analyzing binaries.



Those familiar with Radare2 might recognize that its plugin infrastructure is well-suited to adding architectures already supported by `binutils`. Even if you enjoy leafing through actual pages of objdump output, consider the added value of Radare2's visual mode with colorized output, call graphs, integrated hex editor, and instruction emulation.

Implementing support for a new target architecture isn't as difficult as it might sound. The existing in-tree `nios2` arch support served as a convenient reference and starting point for implementing a `nios` arch plugin. After painstakingly modernizing the relevant code for contemporary compilers from the vintage `binutils` release, it was a quick process to write the required wrapper to hand off a byte sequence for disassembly.

Although this article only covers disassembly, complete target plugins implement an assembler, disassembler, code analysis, and a representation of each opcode using the Evaluable Strings Intermediate Language (ESIL) to enable emulation.

Support for uncommon architectures like Nios tends to end up in the `radare2-extras` repository,[11] otherwise known as the source graveyard, but Radare2 also includes a package manager which can conveniently download and build the plugin from source.

```
$ r2pm -i nios
...
$ r2 -a nios ./hello_world.out
```

As always, build Radare2 from Git master and rebuild often to take advantage of the latest improvements. If you happen to stumble across another rare or otherwise unusual architecture in the course of your hardware adventures, please consider taking a moment to implement your own plugin to keep the architecture alive in all of our hearts and minds.

I hope that you've enjoyed this friendly little guide to Nios, and that you'll keep it handy when reverse engineering firmware from that platform.

[11] `git clone https://github.com/radareorg/radare2-extras.git`


```
1           ;-- strlen:
              0x000809fe      1778          save sp,0x17
3             0x00080a00      1033          mov l0,i0
              0x00080a02      0132          mov g1,l0
5             0x00080a04      0098          pfx hi(0x0)
              0x00080a06      6138          and g1,g3
7             0x00080a08      c17e          skprz g1
          /-< 0x00080a0a      1280          br 0x00080a30
9         |   0x00080a0c      0332          mov g3,l0
          /-> 0x00080a0e      02b0          ldp g2,[l0,0x0]
11        ||  0x00080a10      4130          mov g1,g2
          ||  0x00080a12      f79f          pfx hi(0xfee0)
13        ||  0x00080a14      e437          movi g4,0x1f
          ||  0x00080a16      f79f          pfx hi(0xfee0)
15        ||  0x00080a18      c46f          movhi g4,0x1e
          ||  0x00080a1a      8100          add g1,g4
17        ||  0x00080a1c      413c          andn g1,g2
          ||  0x00080a1e      049c          pfx hi(0x8080)
19        ||  0x00080a20      0234          movi g2,0x0
          ||  0x00080a22      049c          pfx hi(0x8080)
21        ||  0x00080a24      026c          movhi g2,0x0
          ||  0x00080a26      4138          and g1,g2
23        ||  0x00080a28      417f          skprnz g1
          \-< 0x00080a2a      f187          br 0x00080a0e
25        |   0x00080a2c      9004          addi l0,0x4
          |   0x00080a2e      900c          subi l0,0x4
27        \-> 0x00080a30      04b0          ldp g4,[l0,0x0]
              0x00080a32      044e          ext8d g4,l0
29            0x00080a34      447f          skprnz g4
          /-< 0x00080a36      0980          br 0x00080a4a
31        |   0x00080a38      1832          mov i0,l0
          |   0x00080a3a      3004          inc l0
33        /-> 0x00080a3c      01b0          ldp g1,[l0,0x0]
          ||  0x00080a3e      014e          ext8d g1,l0
35        ||  0x00080a40      c17e          skprz g1
          \-< 0x00080a42      fc87          br 0x00080a3c
37        |   0x00080a44      3004          inc l0
          |   0x00080a46      300c          dec l0
39        |   0x00080a48      1832          mov i0,l0
          \-> 0x00080a4a      7808          sub i0,g3
41            0x00080a4c      df7f          ret
              0x00080a4e      a07d          restore
```

Disassembly of `strlen` on Nios.

```
static int disassemble(RAsm *a, RAsmOp *op, const ut8 *buf, int len) {
  if (len < 2) {
    return -1;
  }

  buf_global = &op->buf_asm;
  memcpy(bytes, buf, 2);

  struct disassemble_info info = {0};

  info.disassembler_options = "";
  info.mach = a->bits == 16 ? MACH_NIOS16 : MACH_NIOS32;
  info.buffer = bytes;
  info.read_memory_func = &nios_buffer_read_memory;
  info.symbol_at_address_func = &nios_symbol_at_address;
  info.memory_error_func = &nios_memory_error;
  info.print_address_func = &nios_print_address;
  info.endian = !a->big_endian;
  info.fprintf_func = &nios_fprintf;
  info.stream = stdout;

  op->size = print_insn_nios((bfd_vma) a->pc, &info);

  if (op->size == -1) {
    r_strbuf_set(&op->buf_asm, " (data)");
  }

  return op->size;
}
```

Radare2 plugin for disassembling Nios.